UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

ESTABELECIMENTO DE ROTAS PARA AR.DRONE UTILIZANDO DELPHI 10 SEATTLE

RAFAEL RONALDO RAHN

BLUMENAU
2016

RAFAEL RONALDO RAHN

ESTABELECIMENTO DE ROTAS PARA AR.DRONE UTILIZANDO DELPHI 10 SEATTLE

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Ciência da Computação do Centro de Ciências Exatas e Naturais da Universidade Regional de Blumenau como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Prof. Mauro Mattos, Doutor - Orientador

BLUMENAU
2016

ESTABELECIMENTO DE ROTAS PARA AR.DRONE

UTILIZANDO DELPHI 10 SEATTLE

Por

RAFAEL RONALDO RAHN

Trabalho de Conclusão de Curso aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II pela banca examinadora formada por:

Presidente:	Prof. Mauro Marcelo Mattos, Dr.Eng. – Orientador, FURB
Membro:	Prof. Roberto Heinzle, Dr. Eng – FURB
Membro:	Prof. Miguel Alexandre Wisintainer, Mestre – FURB

Blumenau, 7 de julho de 2016

AGRADECIMENTOS

À minha esposa, Elaine Raquel Klemz, que de forma especial e carinhosa me deu força e coragem, me apoiando nos momentos de dificuldades.

Ao meu orientador, Mauro Marcelo Mattos pela confiança, apoio e conselhos dados durante o desenvolvimento desse trabalho.

Aos professores Dalton Solano dos Reis e Mauricio Capobianco pela ajuda e conselhos no decorrer do trabalho.

Ao professor Roberto Heinzle, companheiro de caminhada ao longo do curso. Eu posso dizer que a minha formação, inclusive pessoal, não teria sido a mesma sem seus conselhos.

RESUMO

Este trabalho descreve o desenvolvimento de uma aplicação denominada AutoDrone, que tem como principal objetivo a criação e o gerenciamento de rotas automatizadas para o ArDrone 2.0 da Parrot e também a posterior condução do drone por estes pontos pré-configurados. O desenvolvimento do aplicativo foi realizado utilizando a ferramenta Delphi 10 Seattle da Embarcadero e o componente TArDrone para a comunicação com o drone. Também foi utilizada a biblioteca GMLib para a integração com o Google Maps. Foram implementados algoritmos para permitir a seleção dos dados que o drone deverá fornecer ao aplicativo de tal forma a viabilizar a análise dos mesmos e o acompanhamento da execução da rota. As ferramentas e componentes utilizados se mostraram adequados e facilitaram o processo de desenvolvimento. Considerando a limitação técnica na precisão do GPS do drone, obteve-se êxito após a realização de vários cenários de testes. A aplicação foi capaz de conduzir o drone pela rota informada, assim como executar a ação configurada no ponto final e também no controle de sua altitude. Estes dados foram obtidos através de medição da posição do drone após o pouso no ponto final da rota.

Palavras-chave: ArDrone.TArDrone. GMLib. Gerenciamento de rotas em *drones*.

ABSTRACT

This work describes the development of an application called AutoDrone, which has as main objective to enable off-line route creation and subsequent automatically driving the ArDrone 2.0 Parrot device. The application was developed using the Delphi 10 Seattle from Embarcadero and used the TArDrone component for communication with the drone. It was also used to GMLib library for integration with Google Maps. Algorithms have been implemented to allow the selection of data that the drone must provide to the application in such a way to enable its analysis and monitoring of the drone's route. The tools and components used were suitable and facilitated the development process. Considering the technical limitations in the drone GPS accuracy, we obtained success after conducting several test scenarios. The application was able to drive the drone through a specified route, as well as to perform some previously specified action in selected points of the route.

Keywords: ArDrone.TArDrone. GMLib. Drone route management.

LISTA DE FIGURAS

Figura 1– Motores do quadricóptero 16
Figura 2 - Movimentos *Yaw*, *Roll* e *Pitch* 17
Figura 3 – Movimentação do quadricóptero 17
Figura 4 – AR.Drone 2.0 Parrot 18
Figura 5 – Mapa com marcadores 23
Figura 6 – Constelação de satélites 24
Figura 7 – Modelo em camadas 27
Figura 8 – Interface Java 28
Figura 9 – Controle remoto 28
Figura 10 – Casos de uso 31
Figura 11 – Principais atividades para definir uma rota 32
Figura 12 – Atividades para gravar uma rota 33
Figura 13 – Atividades para carregar uma rota 34
Figura 14 – Atividades para executar uma rota 35
Figura 15 – Atividades para configurar o aplicativo 36
Figura 16 – Atividades para excluir uma rota 37
Figura 17 – Instalação `TArDrone` 38
Figura 18 – Configuração library path `TArDrone` 39
Figura 19 – Instalação `GmLib` 40
Figura 20 – Configuração library path `GmLib` 40
Figura 21 – Exemplo de um arquivo de rotas 41
Figura 22 – Guia Simulador 42
Figura 23 – Implementação fórmula de Haversine 43
Figura 24 – Gerenciamento de rotas 44
Figura 25 – Guia Informações do AR.Drone 2.0 45
Figura 26 – Guia Logs do sistema 45
Figura 27 – Guia Configurações 47
Figura 28 – Processo de comunicação 48
Figura 29 – Estrutura da NavDat_Demo 50
Figura 30 – Estrutura da NavDat_Magneto 50
Figura 31 – Estrutura da NavDat_GPS 51

Figura 32 - Processo de caminhamento 53
Figura 33 – Cenário 1 54
Figura 34 – Cenários 2 e 3 56
Figura 35 – Área de precisão 58

LISTA DE QUADROS

Quadro 1 – SendCommand 20
Quadro 2 – Envio de comandos 20
Quadro 3 – Comando de configuração 21
Quadro 4 – Comunicação 21
Quadro 5 – Movimentação 21
Quadro 6 – Carga do mapa 22
Quadro 7 – Criação de pontos no mapa 23
Quadro 8 – Equação de Haversine 25
Quadro 9 – Lista de comandos 29
Quadro 10 – Caso de uso `UC01: Definir rota` 32
Quadro 11 – Caso de uso `UC02: Gravar rota` 33
Quadro 12 – Caso de uso `UC03: Carregar rota` 33
Quadro 13 – Caso de uso `UC04: Executar rota` 34
Quadro 14 – Caso de uso `UC05: Configurar aplicativo` 36
Quadro 15 – Caso de uso `UC06: Excluir rota` 36
Quadro 16 – Configuração do retorno 47
Quadro 17 – Recebimento das informações 49
Quadro 18 – Layout da estrutura 49
Quadro 19 – Quebrando o retorno 52
Quadro 20 - Cálculo da direção 53
Quadro 21 – Estruturas mapeadas 63
Quadro 22 – Matrizes 64
Quadro 23 – Velocidades 64
Quadro 24 – Navdata_demo 65
Quadro 25 – Navdat_time 65
Quadro 26 – Navdat_Raw_Measures 65
Quadro 27 – Navdat_Phys_Measures 66
Quadro 28 – Navdat_Gyros_OffSets 66
Quadro 29 – Navdat_Eules_Angles 66
Quadro 30 – Navdat_References 67
Quadro 31 – Navdat_Trims 67

Quadro 32 – Navdat_RC_References....67
Quadro 33 – Navdat_PWM....68
Quadro 34 – Navdat_Altitude....68
Quadro 35 – Navdat_Vision_Raw....69
Quadro 36 – Navdat_Vision_OF....69
Quadro 37 – Navdat_Vision....69
Quadro 38 – Vision_Perf....69
Quadro 39 – Screen_Point_t....70
Quadro 40 – Navdat_Trackers_Send....70
Quadro 41 – Navdat_Vision_Detect....70
Quadro 42 – Navdat_Watchdog....70
Quadro 43 – ADC_Data_Frame....70
Quadro 44 – Navdat_Video_Stream....71
Quadro 45 – Navdat_Games....71
Quadro 46 – Navdat_Pressure_Raw....71
Quadro 47 – Navdat_Magneto....72
Quadro 48 – TChannels....72
Quadro 49 – TNavData....72
Quadro 50 – Navdat_GPS....73

LISTA DE TABELAS

Tabela 1 – Resultados cenário 1 55
Tabela 2 – Resultados cenário 2 56
Tabela 3 – Resultados cenário 3 57
Tabela 4 – Leitura do GPS 58

LISTA DE ABREVIATURAS E SIGLAS

API - *Application Programming Interface*

GB - Gigabytes

Ghz - Giga Hertz

GLPL - *Lesser General Public License*

GMLib - Google Maps Library

GMS - graus, minutos, segundos

GPS - *Global Positioning System*

IDE - *Integrated development environment*

IP - *Internet Protocol*

RAM - *Random Access Memory*

ROS - *Robot Operating System*

SAIL - *Stanford Artificial Intelligence Laboratory*

SDK - *Software Development Kit*

SSID - *Service Set Identifier*

TCP - *Transmission Control Protocol*

UC - *Use Cases*

UDP - *User Datagram Protocol*

VANTs - Veículos Aéreos Não Tripulados

SUMÁRIO

1 INTRODUÇÃO 14
1.1 OBJETIVOS 15
1.2 ESTRUTURA 15
2 FUNDAMENTAÇÃO TEÓRICA 16
2.1 DINÂMICA DE FUNCIONAMENTO DE UM QUADRICÓPTERO 16
2.2 ARDRONE 2.0 18
2.3 COMPONENTE TARDRONE PARA DELPHI 19
2.3.1 Comandos de configuração 20
2.3.2 Comandos de comunicação 21
2.3.3 Comandos de movimentação 21
2.4 COMPONENTE GOOGLEMAPS PARA DELPHI 22
2.5 GPS E SEU FUNCIONAMENTO 23
2.6 FÓRMULA DE HAVERSINE 25
2.7 TRABALHOS CORRELATOS 25
3 DESENVOLVIMENTO 30
3.1 REQUISITOS 30
3.2 ESPECIFICAÇÃO 30
3.2.1 Casos de uso 30
3.3 IMPLEMENTAÇÃO 37
3.3.1 Técnicas e ferramentas utilizadas 37
3.3.2 Arquivo de rotas 40
3.4 OPERACIONALIDADE DA IMPLEMENTAÇÃO 41
3.4.1 Guia Simulador 41
3.4.2 Guia Gerenciamento de rotas 44
3.4.3 Guia Informações do AR.Drone 2.0 44
3.4.4 Guia Logs do sistema 45
3.5 GUIA CONFIGURAÇÕES 45
3.6 OBTENÇÃO DOS DADOS 47
3.6.1 Interpretação dos dados obtidos 49
3.6.2 Cálculo da direção 52
3.7 RESULTADOS E DISCUSSÕES 54

3.7.1 Cenário 1 54
3.7.2 Cenário 2 55
3.7.3 Cenário 3 56
3.7.4 Considerações 57
4 CONCLUSÕES 59
4.1 EXTENSÕES 59
APÊNDICE A – ESTRUTURAS MAPEADAS PARA DELPHI. 63

1 INTRODUÇÃO

Nos últimos tempos a utilização de Veículos Aéreos Não Tripulados (VANTs) vem apresentando uma tendência de crescimento. Inicialmente foram criados com propósitos militares para serem empregados em ações de espionagem, patrulhamento e apoio em artilharia. No entanto, na última década registrou-se o aumento do uso civil dos *drones*. (ALBUQUERQUE, 2013).

Varella (2014) explica que *drones* vem invadindo o universo empresarial e vão influenciar diversos setores, do comércio global partindo da logística, passando pela área de prestação de serviços. No ano de 2013 os valores movimentados pelo mercado mundial de *drones* foram superiores a cinco bilhões de dólares e em dez anos estes valores devem ultrapassar os onze bilhões de dólares (VARELLA, 2014).

Uma outra área que vem liderando a adoção dos *drones* no Brasil onde atuam como ferramenta essencial, é a área de agricultura de precisão. Sua utilização vai desde a captação de fotografias do plantio para avaliação até no auxílio do acompanhamento do desenvolvimento da floresta, por meio de comparação de imagens obtidas de diferentes épocas (VARELLA, 2014).

Conforme Afonso (2014), um *drone* também é capaz de resolver tarefas específicas como vigilância de tráfego, vigilância meteorológica, transporte de cargas específicas ou de resolver tarefas onde a presença de um piloto humano pode ser inviável ou perigosa. O controle de um *drone* é realizado através de um controle remoto ou até um aplicativo rodando em um *smartphone* ou *tablet*, ou seja, existe uma dependência da constante interação humana.

No entanto, esta tarefa pode ser automatizada e para isto é necessária a utilização de uma ferramenta onde seja possível programar as atividades do *drone* fazendo com que o controle da atuação do mesmo seja menos dependente da perícia do operador. Esta automatização requer conhecimento de linguagens de programação de computadores, além de conhecimentos sobre a *Application Programming Interface* (API) do *drone*, o que acaba limitando o escopo às pessoas com este conhecimento técnico.

Visando simplificar o processo de automatização das atividades de estabelecimento de rotas e tarefas para *drones,* propõe-se o desenvolvimento de uma ferramenta que possibilite o controle e programação do mesmo. Especificamente pretende-se utilizar como protótipo o AR Drone 2.0 da empresa Parrot. Através de uma interface visual, o usuário poderá importar um mapa de um ambiente e com o auxílio de marcadores, informar um caminho e ações que o *drone* deverá executar sobre aquele ambiente. Por exemplo, uma tarefa de programação neste

contexto poderia compreender: decolar o *drone*, alinhar-se ao norte, movimentar-se por 10 metros para frente, girar para a esquerda, registrar uma fotografia e retornar ao ponto de origem.

1.1 OBJETIVOS

Este trabalho tem como objetivo disponibilizar um aplicativo que permita a programação de rotas e ações para o ARDRONE 2.0 da empresa Parrot.

Os objetivos específicos do trabalho são:

a) desenvolver um aplicativo de suporte ao estabelecimento de rotas;

b) desenvolver um conjunto de cenários de teste para validar o aplicativo.

1.2 ESTRUTURA

O trabalho desenvolvido está organizado em quatro capítulos. O capítulo 2 apresenta a fundamentação teórica, proporcionando embasamento teórico para compreensão do trabalho. O capítulo 3 apresenta o desenvolvimento da aplicação AutoDrone utilizando Delphi 10 Seattle e a descrição das implementações da aplicação utilizando diagramas de caso de uso, de atividade e sequência. No capítulo 4 são apresentadas as conclusões e as extensões sugeridas.

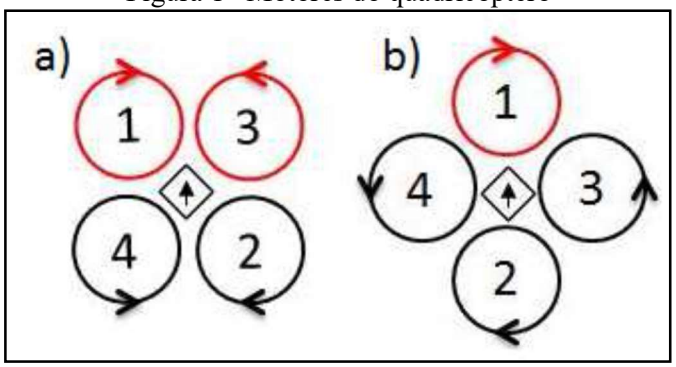
2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo está organizado em quatro seções. Na seção 2.1 será apresentada a dinâmica de funcionamento de um quadricóptero. Na seção 2.2 será apresentada especificamente o AR.Drone 2.0. Na seção 2.3 será apresentado o componente TARDrone, na seção 2.4 será apresentada a biblioteca de componentes para integração com o Google Maps. Na seção 2.5 serão apresentados os conceitos de geolocalização, em seguida na seção 2.6 será apresentada a fórmula de Haversine e na seção 2.7 serão apresentados os trabalhos correlatos à aplicação proposta.

2.1 DINÂMICA DE FUNCIONAMENTO DE UM QUADRICÓPTERO

Um quadrirotor ou quadricóptero é composto basicamente de quatro propulsores organizados em forma de cruz, onde cada uma das hélices é responsável por uma parcela do empuxo total necessário para decolar e sustentar o quadricóptero (PFEIFER, 2013). Para que o quadricóptero se mantenha estável, cada motor tem uma rotação em um sentido específico, de forma que o motor sempre gire no sentido contrário ao seu adjacente (Figura 1– Motores do quadricóptero), e o sistema de voo se dá pelo ajuste da velocidade de cada motor (NASCIMENTO, 2011).

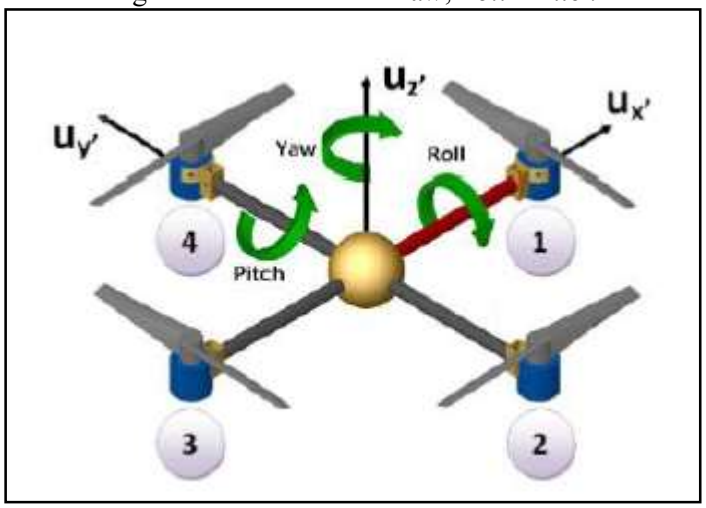
Figura 1– Motores do quadricóptero



Fonte: Nascimento (2011).

O controle dos movimentos é realizado através dos comandos *throttle* (aceleração) que é o movimento vertical, *yaw* (guinada) que corresponde ao movimento de rotação no eixo Z, r*oll* (rolagem) é o movimento de rotação no eixo X, e *Pitch* (arfagem) é o movimento de rotação no eixo Y. A Figura 2 caracteriza os quatro propulsores e os três movimentos citados (NASCIMENTO, 2011).

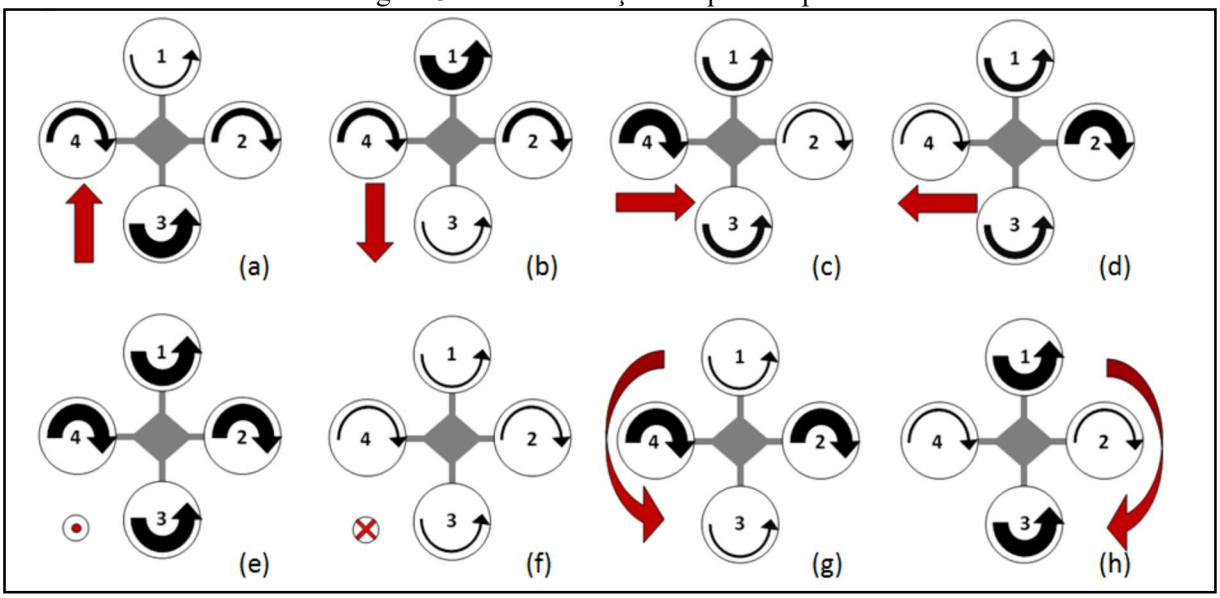
Figura 2 - Movimentos *Yaw*, *Roll* e *Pitch*



Fonte: Pfeifer (2013).

A Figura 3 apresenta os possíveis movimentos. Nela, a largura da seta é proporcional à velocidade dos rotores, também é possível observar que os rotores sempre giram no sentido contrário do adjacente. Fixando a velocidade dos rotores 2 e 4 e aumentando a velocidade do rotor 3 em relação ao 1, o quadrimotor se movimenta para frente, (Figura 3 (a)). Analogamente no item (b) aumentando a velocidade do rotor 1 em relação do 3, obtém-se o movimento para trás. Os movimentos para esquerda e direita ocorrem quando a velocidade dos rotores 1 e 3 permanece a mesma existe a variação dos rotores 2 e 4, figura 3 (c) e (d) (SÁ, 2012).

Figura 3 – Movimentação do quadricóptero



Fonte: Sá (2013).

Ainda na Figura 3 pode-se observar que quando há aumento ou diminuição de velocidade em todos os rotores obtém-se deslocamentos verticais (e) e (f). Quando é alterada a velocidade de dois rotores no mesmo eixo, obtém-se um torque em torno do eixo gerando uma

aceleração angular, por consequência girando o *drone* no sentido horário ou anti-horário, figura 3 (g) e (h) (SÁ, 2012).

2.2 ARDRONE 2.0

AR.Drone 2.0 é um quadricóptero (Figura 4) desenvolvido pela empresa francesa Parrot, que pode ser controlado por computadores ou *smartphones* através de uma conexão *wifi* em modo *ad-hoc* (ponto a ponto) aberta, e operando nas bandas b/g/n (PARROT, 2014). O dispositivo tornou-se conhecido a partir do seu lançamento em virtude do baixo custo e vem sendo utilizado tanto em pesquisas científicas como em aplicações comerciais ou como lazer (AFONSO, 2014).

Figura 4 – AR.Drone 2.0 Parrot

Fonte: Parrot (2014).

Os recursos disponíveis no AR.Drone 2.0 são (PARROT, 2014):

a) Câmera horizontal de 1280 x 720 (720p HD), câmera vertical 320 x 240 (QVGA), acelerômetro nos três eixos para medir a força aplicada ao *drone*, (o qual informa sua inclinação num determinado eixo);
b) Giroscópios nos três eixos para obter a direção do *drone*;
c) Sensor de pressão para maior estabilidade em voos verticais;
d) Ultrassom utilizado para medir a altura do *drone*;
e) Magnetômetro nos 3 eixos (um instrumento utilizado para medir a intensidade e direção de campos magnéticos tendo um funcionamento similar ao de uma bússola 3D);
f) Kernel Linux 2.6.32;
g) Processador ARM Cortex A8 32-bit de 1GHz, memória RAM 1GB e *Wifi* nas

bandas b/g/n.

Além disso, é disponibilizado um recurso chamado *Absolute Control,* o qual permite fazer uso do magnetômetro para que o referencial de controle do quadricóptero seja o ponto de vista do usuário, isto quer dizer que independente de qual a posição do *drone,* o sistema de coordenadas sempre é baseado pela posição do piloto.

Além dos recursos citados, o AR.Drone também conta com algumas vantagens relacionadas ao desenvolvimento de software para que seja possível controlá-lo (AFONSO, 2014):

a) código fonte e *Software Development Kit* (SDK) disponíveis: a Parrot disponibiliza o código fonte e o SDK, que facilita o trabalho do desenvolvedor;
b) documentação: possui documentação completa sobre o SDK;
c) baixo custo: pode-se considerar o custo do AR.Drone relativamente baixo para um quadricóptero com o conjunto de sensores disponíveis.

A comunicação com o AR. Drone 2.0, tanto para controle como para configuração é via uma conexão *wi-fi* criada pelo *drone*, onde ele fornece quatro portas para comunicação, sendo elas:

a) porta 5556 – porta *User Datagram Protocol* (UDP) utilizada para envio de comandos para o drone;
b) porta 5554 – porta UDP utilizada pelo drone para envio dos dados de navegação;
c) porta 5555 – porta UDP utilizada pelo drone para envio de streaming de vídeo;
d) porta 5559 – porta *Transmission Control Protocol* (TCP) utilizada para envio de comandos críticos que não podem ser perdidos, normalmente utilizado para configuração.

2.3 COMPONENTE TARDRONE PARA DELPHI

O componente TARDrone, desenvolvido por McKeeth (2014), fornece uma interface de controle para o AR.Drone 2.0, o qual implementa os comandos básicos de movimentação (DEVELOPER, 2014). O Quadro 1 apresenta a implementação do método de envio de comandos ao AR.Drone, onde é utilizada uma comunicação do tipo *User Datagram Protocol* (UDP) (MCKEETH, 2014).

Quadro 1 – SendCommand

```
procedure TARDrone.SendCommand(cmd, arg: string);
var
  full: string;
begin
  if not udp.Active then Connect;

  full := Format('%s=%d,%s' + Chr(13), [Cmd, Seq, arg]);
  Seq := Seq + 1;
  udp.Send(full);
end;
```

Fonte: McKeeth (2014).

No Quadro 2 é apresentada a utilização da *procedure* `SendCommand` do componente, a qual envia uma sequência de comandos para o *drone* decolar, aterrissar, restrição de altitude e parada de movimentos (MCKEETH, 2014).

Quadro 2 – Envio de comandos

```
SendCommand('AT*REF','290718208'); // Takeoff
SendCommand('AT*REF','290717696'); // Land
SendCommand('AT*CONFIG', '"control:altitude_max","10000"'); // unlimited
SendCommand('AT*CONFIG', '"control:altitude_max","5000"'); // restrituded
SendCommand('AT*PCMD','1,0,0,0,0'); // Hover (stop movement)
```

Fonte: McKeeth (2014).

Além disso, é possível enviar comandos para pouso, decolagem, emergência e um comando de envio de comandos genéricos de acordo com a *Application Programming Interface* (API) do AR. Drone 2.0. O componente é de código fonte aberto e está disponível para as plataformas Delphi XE 6, Appmethod, C++Builder and RAD Studio e continua recebendo atualizações (MCKEETH, 2014). Os comandos fornecidos pelo componente TARDrone podem ser classificados nos seguintes grupos: configuração, comunicação e movimentação.

2.3.1 Comandos de configuração

Os comandos de configuração compreendem: `RenameSSID`, `RenameDrone`, `RestrictAltitude` e `UnlimitedAltitude`. `RenameSSID` que tem a função de renomear o *Service Set IDentifier* (SSID) (é uma *string* (texto) de até 32 caracteres que identifica cada rede sem fio). O comando `RenameDrone` permite renomear o *drone*. O comando de restrição de altitude `RestrictAltitude` serve para configurar a altitude máxima que o *drone* irá atingir e comando `UnlimitedAltitude` serve para remover a restrição de altitude. O Quadro 3 apresenta a uma sequência de comandos onde é restringida a altitude máxima para o *drone* subir em dois metros, é renomeado o nome do *drone* para “DroneTCC” e é renomeado a identificação da rede para “RedeArDroneTCC”.

Quadro 3 – Comando de configuração

```
begin
  ARDrone.RestrictAltitude(2000);
  ARDrone.RenameDrone('DroneTCC');
  ARDrone.RenameSSID('RedeArDroneTCC');
end;
```

2.3.2 Comandos de comunicação

Para comunicação do controlador com o *drone* estão disponíveis os comandos de início e fim do processo de comunicação com o *drone*. Esta comunicação é realizada via UDP. O Quadro 4 apresenta um exemplo de comunicação com o AR.Drone utilizando o componente, onde é configurada porta de comunicação informando *Internet Protocol* (IP) e a porta do AR.Drone (MCKEETH, 2014).

Quadro 4 – Comunicação

```
udp := TIdUDPClient.Create(nil);
udp.Host := '192.168.1.1';
udp.Port := 5556;
seq := 1;
```

Fonte: McKeeth (2014).

2.3.3 Comandos de movimentação

Para movimentação estão disponíveis os comandos para mover para frente, para trás, para direita, para esquerda, para cima e para baixo, de rotação do *drone* no sentido horário e anti-horário. O Quadro 5 apresenta um exemplo de código com os comandos básicos de movimentação e rotação (MCKEETH, 2014).

Quadro 5 – Movimentação

```
begin
  ARDrone.MoveForward(1);

  ARDrone.MoveBackward(1);

  ARDrone.MoveRight(1);

  ARDrone.MoveLeft(1);

  ARDrone.RotateCW(1);

  ARDrone.RotateCCW(1);

end;
```

Fonte: McKeeth (2014).

2.4 COMPONENTE GOOGLEMAPS PARA DELPHI

O Google Maps Library (GMLib) é um conjunto de componentes, disponíveis para Delphi e C++ Builder, que encapsulam a API do Google Maps. Eles possuem recursos para tratar marcadores, polígonos, retângulos e polilinhas. Estes componentes estão disponíveis através da licença *Lesser General Public License* (GLPL) (SANTOS, 2013).

Os componentes disponíveis para utilização são:

a) TGMMap: é o principal componente especializado para o TWebBrowser e é responsável pela conexão com o Google Maps, os demais devem estar ligados neste;
b) TGMMarker: componente responsável pelo gerenciamento dos marcadores no mapa;
c) TGMInfoWindow: componente utilizado para exibir um balão informativo;
d) TGMPolyline: componente utilizado para adicionar polilinhas no mapa;
e) TGMPolygon: componente utilizado para adicionar polígonos no mapa;
f) TGMRectangle: componente utilizado para adicionar retângulos no mapa;
g) TGMCircle: utilizado para adicionar círculos no mapa;
h) TGMGeoCode: componente que disponibiliza um serviço para conversão de um endereço para uma coordenada geográfica;
i) TGMDirection: componente responsável por fornecer o cálculo de rotas entre dois pontos;
j) TGMElevation: componente responsável por disponibilizar o cálculo de elevação do terreno;
k) TGMGroundOverlay: componente que permite sobrepor imagens no mapa.

Para que o mapa seja processado pela aplicação e exibido no TWebBrowser ligado ao componente GMMap, deve-se executar o comando `DoMap` no evento `AfterPageLoaded` do GMMap. Conforme é demostrado no Quadro 6

Quadro 6 – Carga do mapa

```
procedure TfrmPrincipal.GMMap1AfterPageLoaded(Sender: TObject; First: Boolean);
begin
  GMMap1.DoMap;
end;
```

Fonte: Santos (2013).

No Quadro 7 é demonstrado a utilização do componente GMGeoCode onde é chamado o comando `GeoCode` e passado como parâmetro um endereço do local. O componente GMGeoCod realiza a conversão deste endereço para um ponto geográfico e cria um marcador

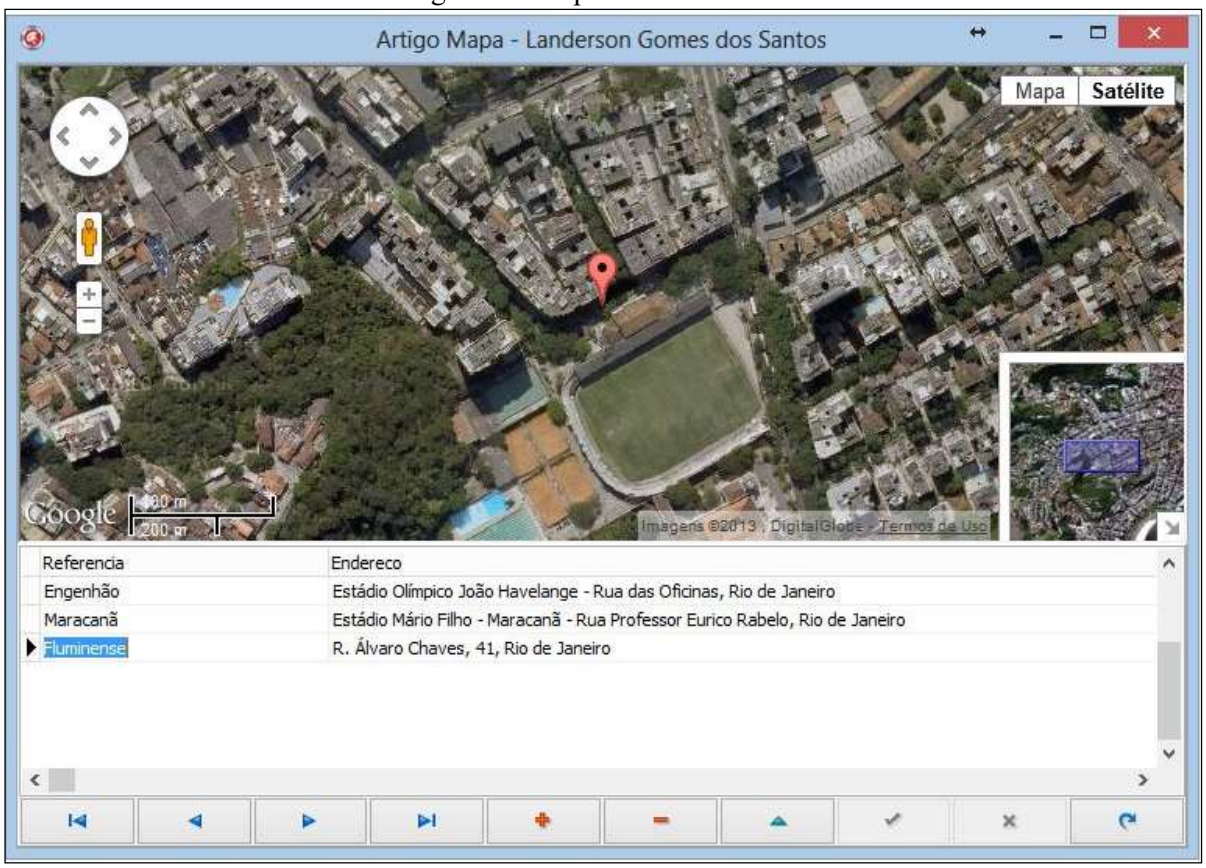
no mapa para este ponto. Em seguida é utilizado o comando `CenterMapToMarker` do componente GMMarker, para posicionar o centro do mapa no TWebBroser. O resultado final com o mapa centralizado e o marcador pode ser visto na Figura 5 (SANTOS, 2013).

Quadro 7 – Criação de pontos no mapa

```
procedure TfrmPrincipal.DBGrid1DblClick(Sender: TObject);
begin
  GMMap1.Active := True;
  if DataSource1.DataSet.RecordCount > 0 then
    begin
      GMGeoCode1.Geocode(ClientDataSet1Endereco.AsString);
      GMMarker1.Items[DataSource1.DataSet.Recno].CenterMapToMarker;
    end;
end;
```

Fonte: Santos (2013).

Figura 5 – Mapa com marcadores



Fonte: Santos (2013).

2.5 GPS E SEU FUNCIONAMENTO

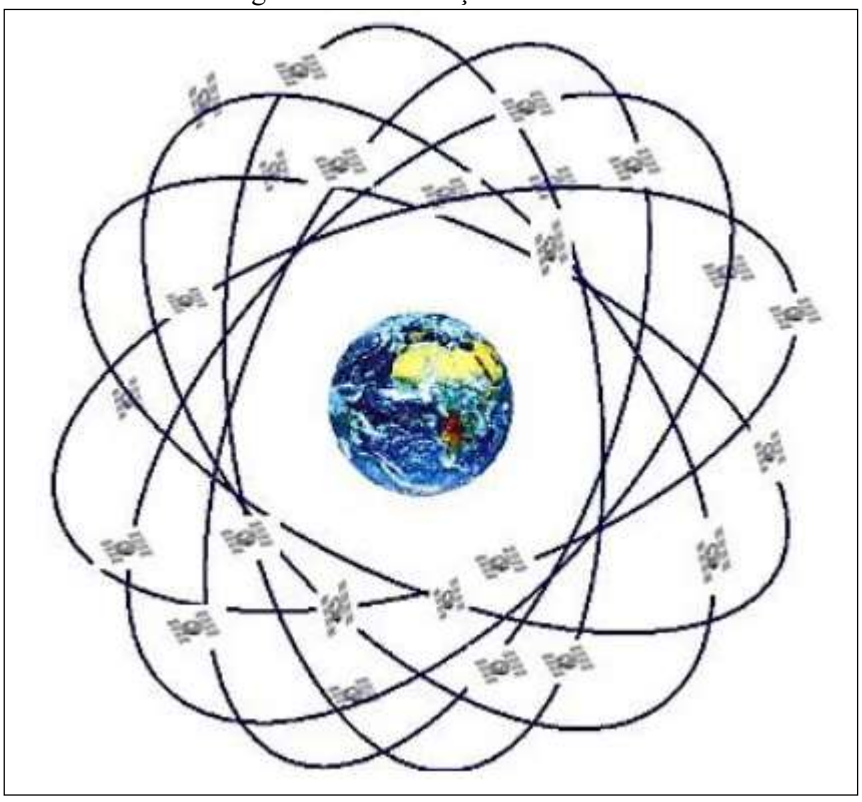
Durante a década de 1970, surgiu o Global Positioning System (GPS), desenvolvido pelo Departamento de Defesa (DoD) dos Estados Unidos da América (EUA). Foi o sucessor do sistema TRANSIT, primeiro sistema de navegação por satélite também conhecido por Navy Navigation Satellite System, que foi amplamente utilizado pela marinha dos EUA, assim como

pela comunidade civil, mas que se tornou obsoleto com a entrada em funcionamento do GPS a 17 de julho de 1995 (SANTOS, 2015).

Segundo Souza (2015), o GPS é um sistema que permite um posicionamento global contínuo, foi inicialmente desenvolvido para atender as necessidades militares dos EUA. Em seguida passou a ser disponibilizado para uso de civis, porém com precisão menor do que disponível para operações militares.

O GPS é constituído por três segmentos, espacial (constelação de satélites), de controle (militares) e usuários (receptores). Referente ao segmento espacial, inicialmente previsto um total de 24 satélites (Figura 6) orbitando a Terra a aproximadamente 20.200 km acima do nível do mar. Distribuídos em 6 órbitas garantindo que quatro satélites estejam sempre visíveis em qualquer momento e qualquer ponto do globo. Atualmente o sistema conta com total de 32 satélites (SANTOS, 2015).

Figura 6 – Constelação de satélites



Fonte: Souza (2015).

Cada satélite do GPS transmite por rádio um padrão fixado que é recebido pelos receptores da Terra. Todos os satélites enviam seus sinais ao mesmo tempo, possibilitando assim o receptor avaliar o lapso de tempo entre emissão e recepção (MOI; MOREIRA; GEREMIA, 2014).

O segmento de controle é caracterizado por uma rede de estações de rastreio e uma de controle principal localizada em Colorado Springs nos EUA. No segmento de usuário se

encontram todos utilizadores civis e militares que possuem o equipamento para recepção do sinal GPS (SANTOS, 2015).

2.6 FÓRMULA DE HAVERSINE

Segundo Sanches (2012), o cálculo da distância exata entre dois pontos na Terra é algo complexo, pois deve considerar o relevo do terreno. No entanto, pode-se obter uma precisão aceitável se considerarmos o formato da Terra como sendo uma esfera. Neste caso pode-se utilizar a equação de Haversive exibida no Quadro 8.

Quadro 8 – Equação de Haversine

$$d = 2R\ arcsen\sqrt{sen^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos(\phi_1)\cos(\phi_2)sen^2\left(\frac{\psi_2 - \psi_1}{2}\right)}$$

Fonte: Sanches (2012).

Esta equação estabelece:

a) d é a distância entre dois pontos da superfície da esfera em quilômetros;
b) R é o raio da esfera em quilômetros;
c) ϕ_1 e ϕ_2 é a latitude dos dois pontos em graus;
d) ψ_1 e ψ_2 é a longitude dos dois pontos em graus.

Em projetos de estabelecimento de rotas, como é o caso do presente projeto, a fórmula de Haversine é utilizada para realizar o cálculo da distância entre dois pontos.

2.7 TRABALHOS CORRELATOS

Entre os trabalhos correlatos destacam-se os de Ros (2007), Portal (2011) e Afonso (2014), pois todos guardam relação com as tecnologias utilizadas ou linha de pesquisa do trabalho proposto.

O primeiro trabalho considerado como correlato é o *Robot Operating System* (ROS), ou Sistema Operacional de Robôs, o qual foi originalmente desenvolvido em 2007 pelo laboratório de inteligência artificial de Stanford (SAIL) com apoio do projeto de IA para robôs de Stanford. Em 2008, o desenvolvimento continuou principalmente em *Willow Garage*, um centro de pesquisa robótica, com a colaboração de mais de 20 instituições em um modelo federativo de desenvolvimento (RAMOS, 2014).

O ROS disponibiliza bibliotecas e ferramentas para auxiliar desenvolvedores de software a criar aplicações robóticas. Ele fornece abstração de *hardware, device* e *drivers,*

bibliotecas, visualizadores, transmissão de mensagens, gerenciamento de pacotes e muito mais (O'KANE, 2013).

Algumas características do ROS é que ele está licenciado sob uma licença livre (a licença BSD) e foi construído a partir do zero para incentivar projetos colaborativos de robótica. Por exemplo, um laboratório pode ter especialistas em mapeamento de ambientes internos, e poderia contribuir com um sistema para a produção de mapas. Outro grupo pode ter especialistas em usar mapas para navegar, e ainda outro grupo pode ter descoberto uma abordagem de visão computacional que funciona bem para o reconhecimento de pequenos objetos em desordem (ROS, 2007).

Para utilizar o ROS é necessário um sistema operacional Ubuntu (existem versões experimentais em Fedora Linux, Mac OS X, e Microsoft Windows). Para o desenvolvimento podem ser usadas diversas linguagens de programação como C++, Python, Lisp, Java, Lua, C# e Ruby. Além disso, ele fornece suporte para uma série de plataformas de robóticas como, por exemplo, AMIGO, AscTec Quadrotor, Intel Edison, Lego NXT, Softbank Pepper e inúmeros outros (ROS, 2007).

Dentre os módulos do ROS pode-se destacar o sistema de mensagem com o robô, sistema de posicionamento, localização e navegação, sistema de diagnósticos e talvez a ferramenta mais conhecida do ROS, RVIZ que fornece visualização tridimensional de vários tipos de dados do sensor.

Já o trabalho de Portal (2011), descreve uma aplicação de piloto automático para o AR Drone 2.0, o qual tem o objetivo de fazer o *drone* cumprir um itinerário pré-definido. O itinerário consiste em uma sequência de marcadores no chão pelos quais o *drone* deve passar.

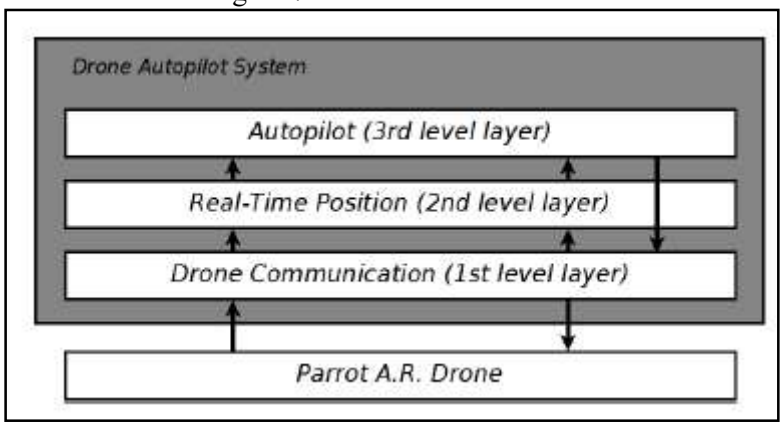
O piloto automático é um *software* que está sendo executado em um computador. O computador recebe os dados capturados pelos sensores do *drone* via uma conexão *wifi*. O computador processa esses dados recebidos do *drone* e envia o próximo comando que o *drone* deve executar.

O *software* deste trabalho foi desenvolvido em linguagem Java. Para a implementação foi utilizado o DiaSpec, que é uma ferramenta de geração de automática de código a partir de um diagrama, a qual foi desenvolvida pela equipe *Phoenix* do laboratório *Institut National de Recherche en Informatique et en Automatique* (INRIA) (PORTAL, 2011).

A abordagem utilizada na construção do piloto automático foi de um modelo de *software* em camadas (PORTAL, 2011). A Figura 7 ilustra este modelo, onde a primeira camada de comunicação recebe os dados enviados pelo *drone* e também é responsável pelo envio dos comandos. A segunda camada, é a camada que realiza continuamente os cálculos de

posicionamento do *drone* no espaço. A terceira camada é o piloto automático que faz uso das informações enviadas pelas outras camadas e através de uma máquina de estados decide o comando a ser enviado ao *drone*.

Figura 7 – Modelo em camadas



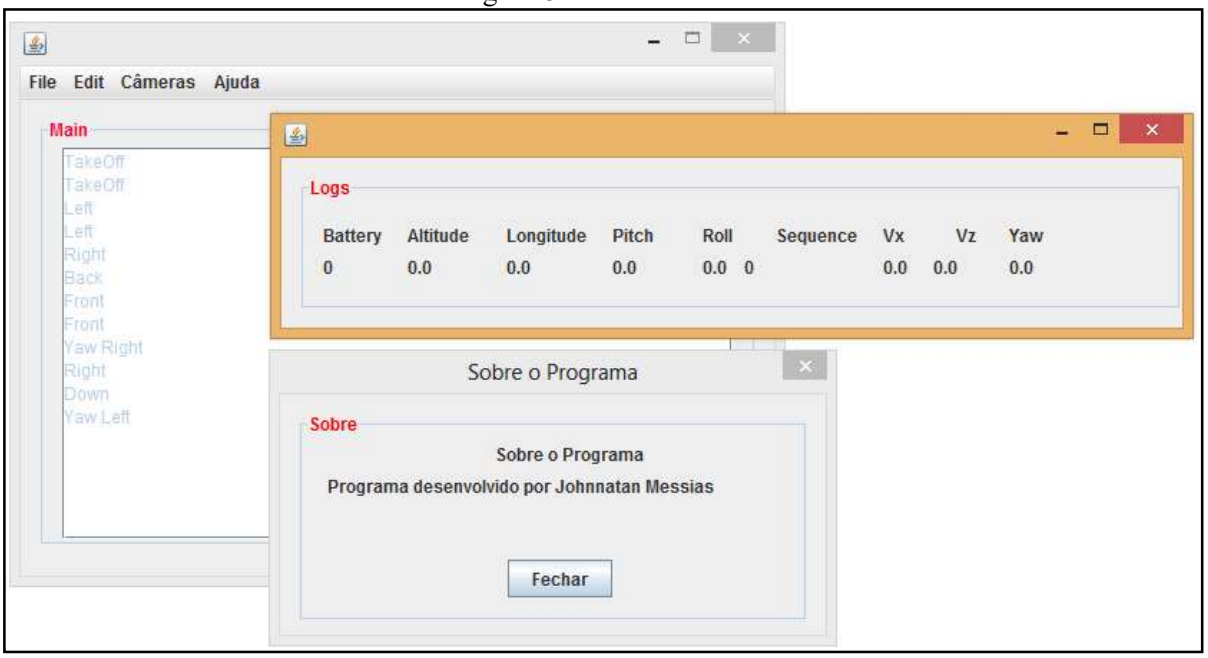
Fonte: Portal (2011).

Já Afonso (2014) apresenta dois aplicativos de voos autônomos para o AR. Drone 2.0, sendo o primeiro desenvolvido em Java e o segundo em Node.js. Em ambos os aplicativos, o usuário pode criar missões de voos para que o *drone* as execute.

Inicialmente a aplicação foi desenvolvida em Java utilizando a API JavaDrone (2014) que é uma API em Java que implementa os protocolos de comunicação do *drone*. Para essa aplicação foi desenvolvida uma interface gráfica (Figura 8), elaborada de maneira bem simples para ser utilizada nos testes de voo da aplicação. Além disso, foram implementadas duas classes, onde uma é responsável pela movimentação do *drone* e outra por obter e armazenar em um arquivo de texto os dados sensoriais coletados pelo *drone* (AFONSO, 2014).

Conforme o autor, após os testes iniciais com o aplicativo desenvolvido em Java foi verificado que desenvolver estratégias de voos autônomos usando a linguagem Java seria trabalhoso uma vez que há ferramentas mais adequadas para esse tipo de tarefa (AFONSO, 2014). Buscou-se então uma nova solução, através de uma linguagem ou *framework* que proporcionasse maior abstração para a realização de voos autônomos (AFONSO, 2014). O segundo aplicativo desenvolvido no *framework* Node.js e fez uso de um controle remoto (Figura 9) e um *kit* arduino onde cada botão representa uma função pré-definida (Quadro 9).

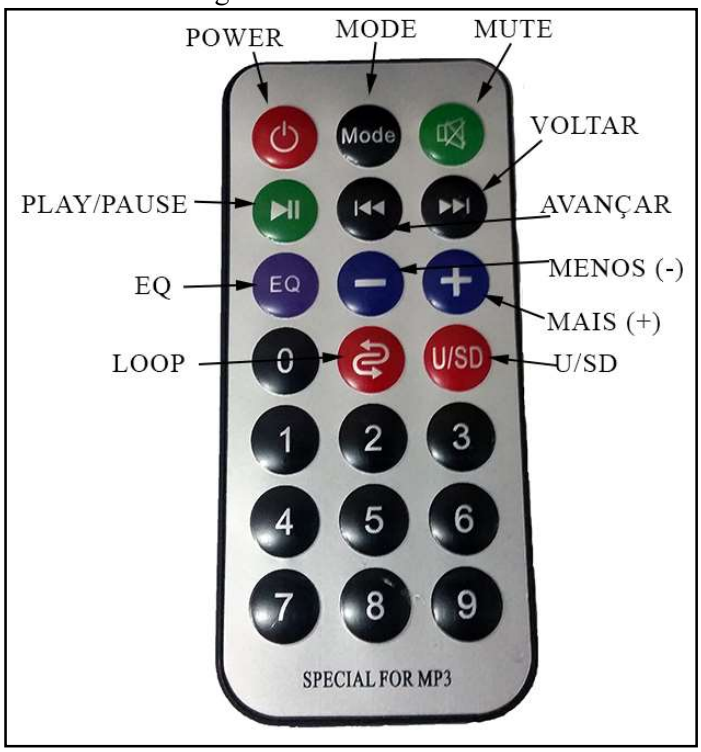
Figura 8 – Interface Java



Fonte: Afonso (2014).

Na figura 9 pode-se observar o controle remoto do com os seus respectivos botões. Do canto superior esquerdo para a direita tem-se, `Power`, `Mute`, `Play/Pause`, `Voltar`, `Avançar`, `Eq`, `Menos (-)`, `Mais (+)`, `0`, `Loop U/SD`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8` e `9`. No Quadro 9 está relacionado cada botão controle remoto com sua função pré-programada no Ar Drone 2.0.

Figura 9 – Controle remoto



Fonte: Afonso (2014).

Quadro 9 – Lista de comandos

Botões	**Funções**	**Botões**	**Funções**
Power	TakeOff/Landing	Mode	-
Mute	Desconectar/Sair	Play/Pause	Subir/Descer
Voltar	Mover para Trás	Avançar	Mover para a Frente
Eq	Executar missão	Menos(-)	Mover Yaw para Esquerda
Mais (+)	Mover Yaw para Direita	0	Teste de comunicação
Loop	Decrementa Angle em 15	U/SD	Incrementa Angle em 15
1	Executa Missão Quadrática	2	Executa Missão Linear
3	Executa Missão Triangular	4	Executa Missão Circular
5	Mover para a esquerda	6	Move para a direita
7	Decrementa Value em 1	8	Incrementa Value em 1
9	Posiciona o drone no ponto de origem	-	-

Fonte: Afonso (2014).

3 DESENVOLVIMENTO

Neste capítulo são apresentadas as etapas de desenvolvimento do aplicativo, bem como os testes efetuados. Na primeira seção são abordados principalmente os requisitos funcionais e não funcionais.

Na segunda seção são apresentados os casos de uso e seus diagramas de atividades utilizando UML (*Unified Modeling Language)*. A terceira seção detalha a implementação do aplicativo, apresentando os principais trechos de código e detalhamentos das funcionalidades das telas. E por último, a quarta seção aborda os resultados obtidos deste trabalho.

3.1 REQUISITOS

Os requisitos para a ferramenta proposta são:

a) permitir criar um mapa de navegação (Requisito Funcional - RF);
b) permitir o estabelecimento de ações em posições específicas do mapa (RF);
c) realizar *upload* das tarefas para o *drone* (RF);
d) disparar o voo autônomo a partir da rota pré-programada (RF);
e) permitir a importação de um mapa 2D com marcadores para ferramenta (RF);
f) permitir que após o mapa importado seja possível desenhar uma rota para o *drone* percorrer (RF);
g) permitir salvar as rotas do mapa (RF);
h) calibrar a rota para corrigir eventuais desvios durante a navegação do *drone* (RF);
i) utilizar a IDE Delphi 10 Seattle (Requisito Não Funcional - RNF);
j) utilizar o Ar.Drone 2.0 (RNF).

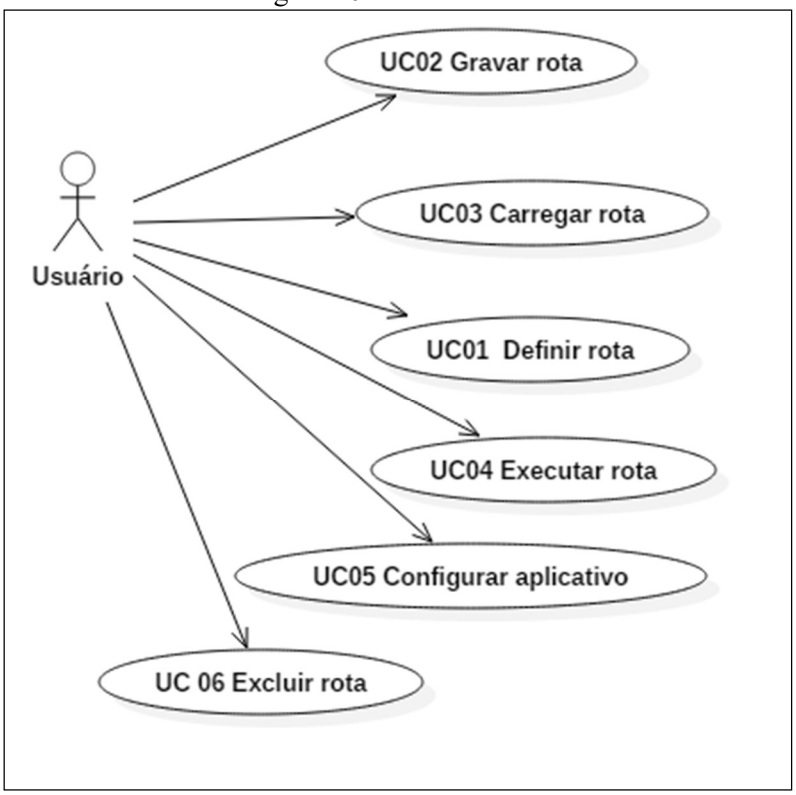
3.2 ESPECIFICAÇÃO

Para a especificação do trabalho foram criados diagramas de casos de uso, de atividades e de sequência com a ferramenta StarUML e utilizado a notação *Unified Modeling Language* (UML). Nas próximas seções as especificações detalhadas são apresentadas.

3.2.1 Casos de uso

Esta seção apresenta os casos de uso que descrevem as funcionalidades do aplicativo. No aplicativo existe apenas um tipo de ator, chamado de `usuário`, que tem disponível as funcionalidades de `criar, gravar, carregar e executar` as rotas além de `configurar` a aplicação, conforme pode ser visto na Figura 10.

Figura 10 – Casos de uso



Para cada caso de uso foi desenvolvido ao menos um cenário, tendo sido realizada a vinculação com os respectivos requisitos funcionais. As descrições dos *Use Cases* (UC), condições, cenários e exceções são detalhados a seguir.

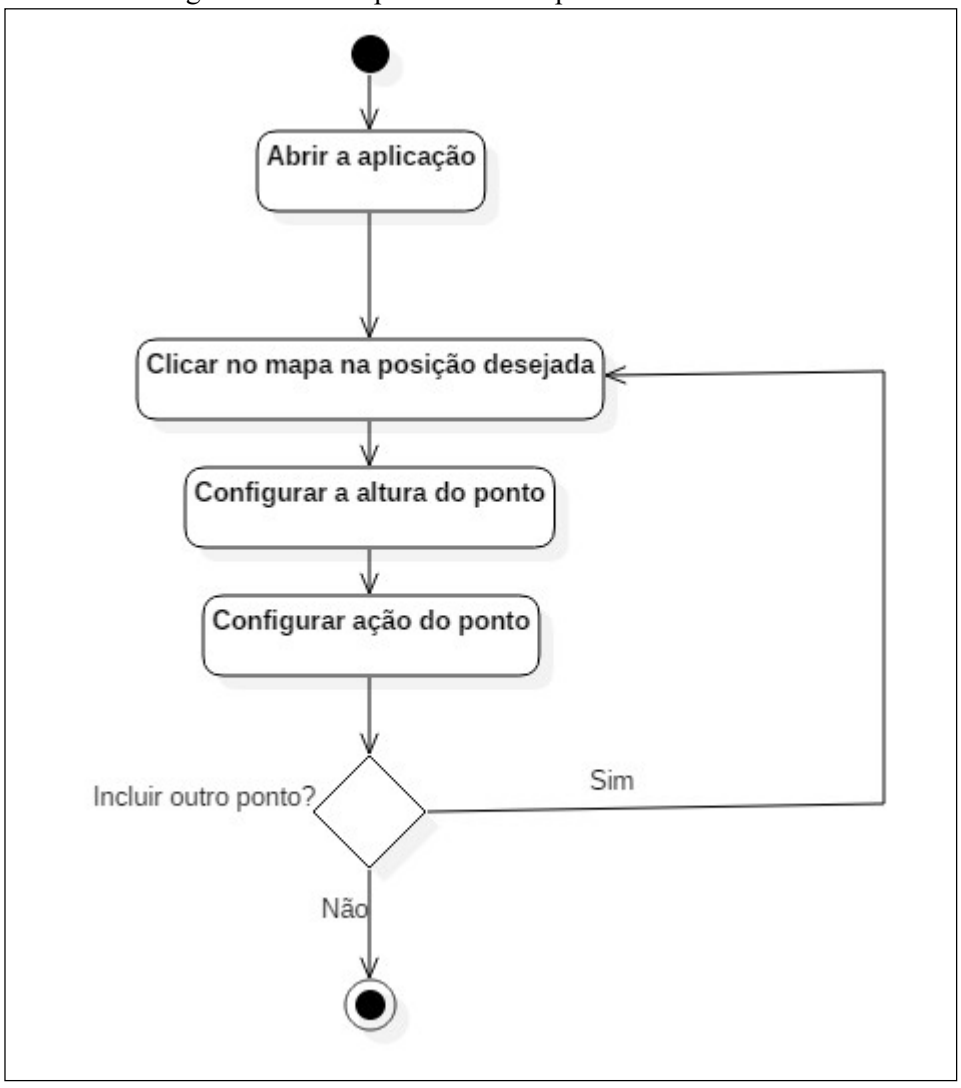
3.2.1.1 Caso de uso: `Definir uma rota`

Abrindo-se a aplicação, uma das primeiras tarefas que devem ser feitas é a definição de uma rota. Rota neste contexto é um conjunto de pontos que o *drone* deverá alcançar e a cada ponto poderá ser configurada uma altura e uma ação a ser executada. No Quadro 10 é apresentado o detalhamento das operações para definição de uma rota, em seguida a Figura 11 mostra o mesmo detalhamento utilizando um diagrama de atividades.

Quadro 10 – Caso de uso UC01: Definir rota

UC01 – Definir rota	
Descrição	Permite que o usuário do AutoDrone crie uma rota dentro da aplicação
Cenário principal	1. Usuário do AutoDrone clica no mapa para definir o ponto. 2. Usuário do AutoDrone configura altura do ponto 3. Usuário do AutroDrone configura a ação que o drone deve executar quando chegar no ponto 4. Repetir os passos 1, 2 e 3 para cada ponto desejado
Pós-Condição	Ter uma rota definida

Figura 11 – Principais atividades para definir uma rota



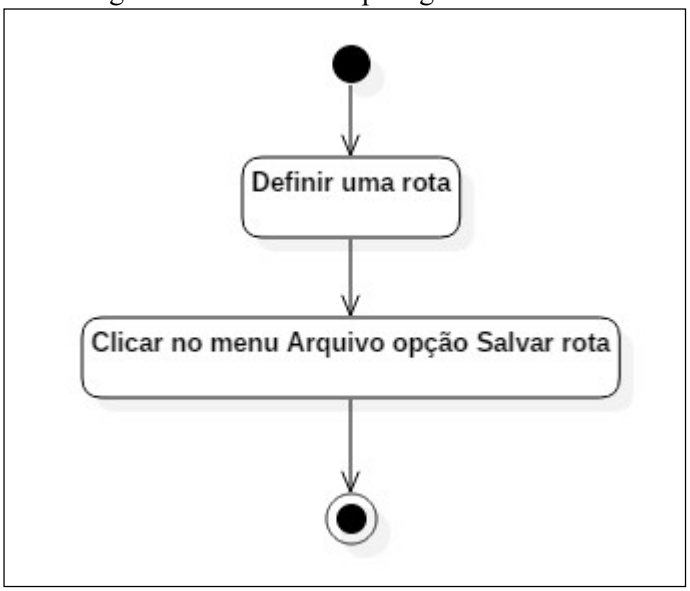
3.2.1.2 Caso de uso: Gravar rota

Após a definição da rota ter sido realizada, a aplicação fornece um recurso para que esta rota seja armazenada. O Quadro 11 e a Figura 12 apresentam o caso de uso e o diagrama de atividades para este procedimento.

Quadro 11 – Caso de uso UC02: Gravar rota

UC02 – Grava rota	
Descrição	Permitir que o usuário grave uma rota já definida
Cenário principal	1. Usuário clica no menu Arquivo na opção Gravar Rota 2. Usuário informa um nome para o arquivo
Pré-Condição	Ter uma rota definida
Pós-Condição	O usuário ter a rota gravada

Figura 12 – Atividades para gravar uma rota



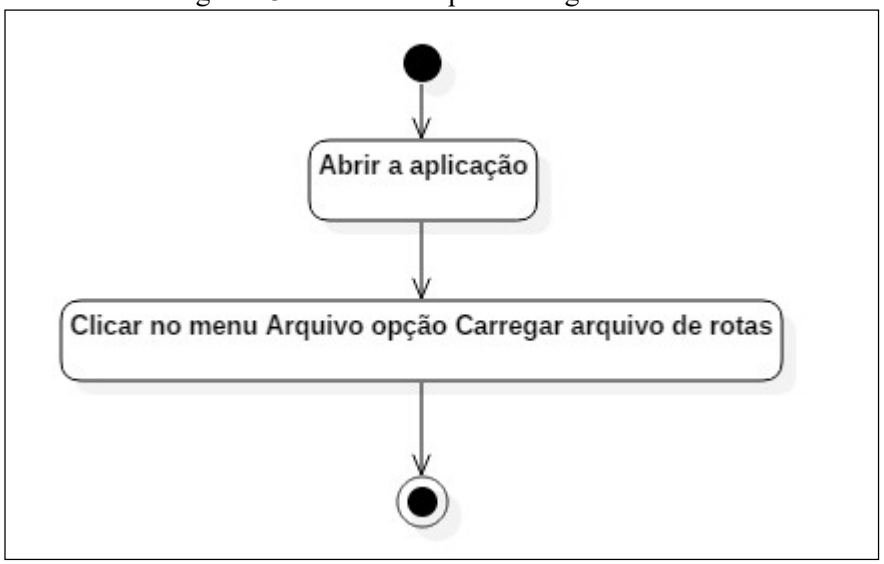
3.2.1.3 Caso de uso: Carregar rota

Após a gravação das rotas é possível carregá-las novamente para a aplicação. O Quadro 12 e a Figura 13 apresentam o caso de uso e o diagrama de atividade para este procedimento.

Quadro 12 – Caso de uso UC03: Carregar rota

UC03 – Carregar rota	
Descrição	Permitir que o usuário abra um arquivo de rotas previamente gravado
Cenário principal	1. Usuário clica no menu Arquivo na opção Carregar arquivo de rotas 2. Usuário seleciona o arquivo
Pré-Condição	Ter algum arquivo de rotas já gravado
Pós-Condição	A rota estar carregada para a aplicação

Figura 13 – Atividades para carregar uma rota



3.2.1.4 Caso de uso: `Executar rota`

Após ter sido definida uma rota ou uma rota ter sido carregada para a aplicação é possível enviar o comando para que o *drone* execute esta rota (Quadro 13). Como pode ser observado na Figura 14, inicialmente é necessário posicionar o *drone* no ponto de partida com a frente alinhada para o norte, em seguida deve-se conectar à rede *wireless* criada pelo Ar.Drone, e depois abre-se se a aplicação e deve ser selecionado uma rota a ser executada.

Quadro 13 – Caso de uso `UC04: Executar rota`

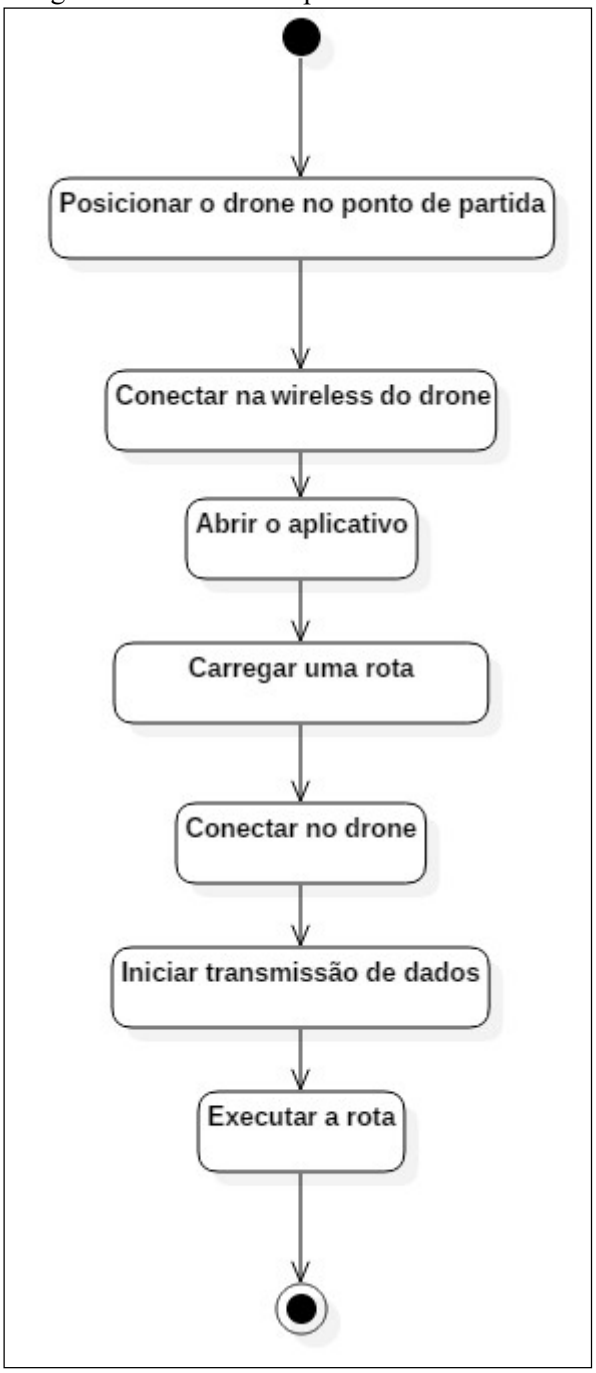
`UC04 – Executar rota`	
`Descrição`	`Permitir que o usuário execute uma rota previamente definida`

`Cenário principal`	`1. Usuário posiciona o drone no ponto de partida com a frente alinhada para o norte`
	`2. Usuário se conecta à rede wireless fornecida pelo drone`
	`3. Usuário carrega uma rota`
	`4. Usuário estabelece conexão com o drone`
	`5. Usuário inicia a transmissão de dados com o drone`
	`6. Usuário dá o comando de levantar voo`
	`7. Usuário dá o comando de mover`

`Pré-Condição`	`Ter uma rota definida`
`Pós-Condição`	`Drone deverá estar posicionado no último ponto da rota`

Após deve-se conectar ao *drone* e iniciar a transmissão de dados, e finalmente pode ser dado início a execução da rota.

Figura 14 – Atividades para executar uma rota



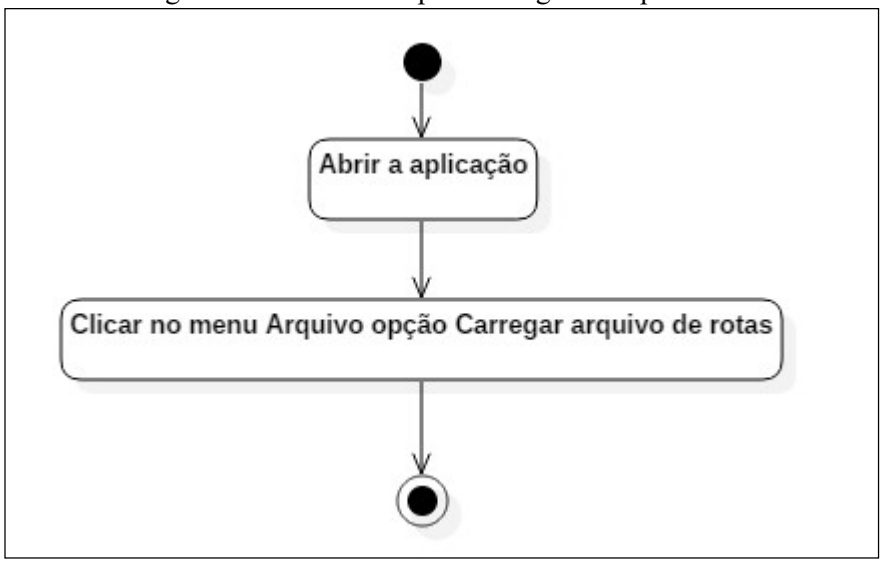
3.2.1.5 Caso de uso: `Configurar aplicativo`

Para que seja possível melhorar a performance e assertividade do *drone* durante o seu caminhamento, é possível alterar estes parâmetros configurando a aplicação. Esta configuração pode inclusive ser realizada durante o voo do *drone* e os novos cálculos levaram em consideração estes novos valores informados. O Quadro 14 e a Figura 15 respectivamente apresentam o caso de uso e o diagrama de atividade para este procedimento.

Quadro 14 – Caso de uso UC05: Configurar aplicativo

UC05 – Configurar aplicativo	
Descrição	Permitir que o usuário configure os parâmetros da aplicação
Cenário principal	1. Usuário informa os valores dos parâmetros
Pós-Condição	A aplicação faz uso dos novos valores informados

Figura 15 – Atividades para configurar o aplicativo



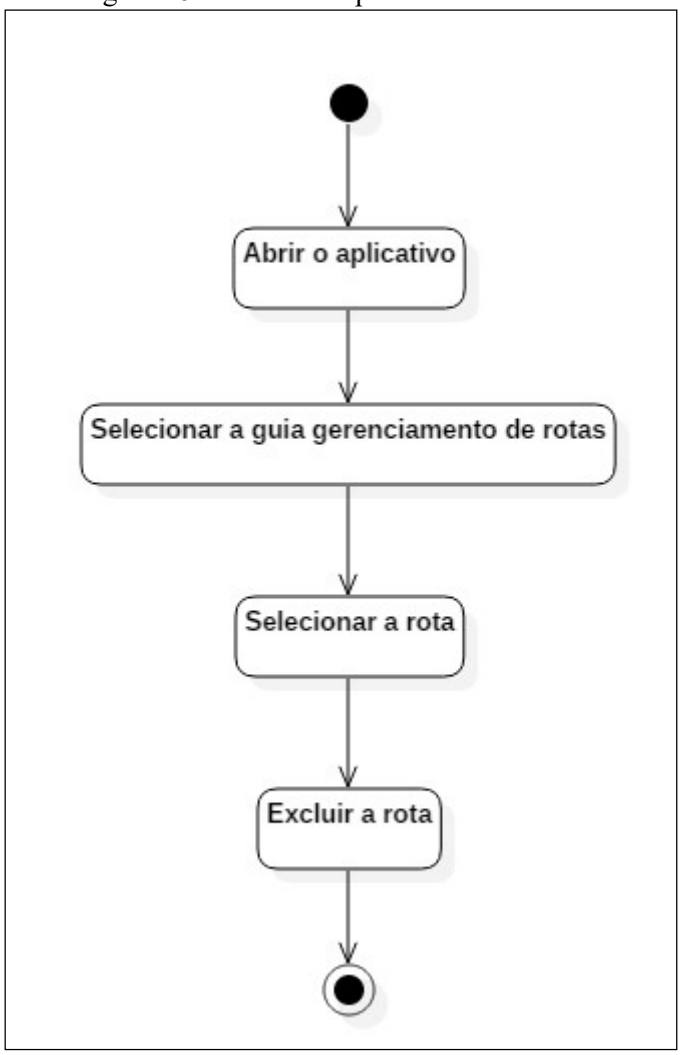
3.2.1.6 Caso de uso: Excluir rota

Caso seja necessário, a aplicação fornece um recurso para a exclusão de rotas previamente gravadas. O Quadro 15 e a Figura 16 apresentam o caso de uso e o diagrama de atividade para este procedimento.

Quadro 15 – Caso de uso UC06: Excluir rota

UC02 – Grava rota	
Descrição	Permitir que o usuário exclua uma rota já gravada
Cenário principal	1. Usuário clica na guia Gerenciamento de rotas 2. Usuário localiza a rota a ser excluída na grid 3. Usuário clica com o botão direito sobre ela e seleciona a opção excluir
Pré-Condição	Ter uma rota gravada
Pós-Condição	A rota selecionada ter sido excluída

Figura 16 – Atividades para excluir uma rota



3.3 IMPLEMENTAÇÃO

A seguir são apresentadas as técnicas, ferramentas e componentes utilizados na implementação da aplicação. Em seguida são detalhadas as telas da aplicação, explicando sua operacionalidade. Também se apresenta o código fonte das rotinas mais importantes da aplicação.

3.3.1 Técnicas e ferramentas utilizadas

Para o desenvolvimento da aplicação AutoDrone, foi utilizada a linguagem Delphi com a *Integrated development environment* (IDE) Delphi 10 Seattle da empresa Embarcadero. Além disso foram utilizados os componentes TArDrone e *Google Maps Library* v1.5.3.

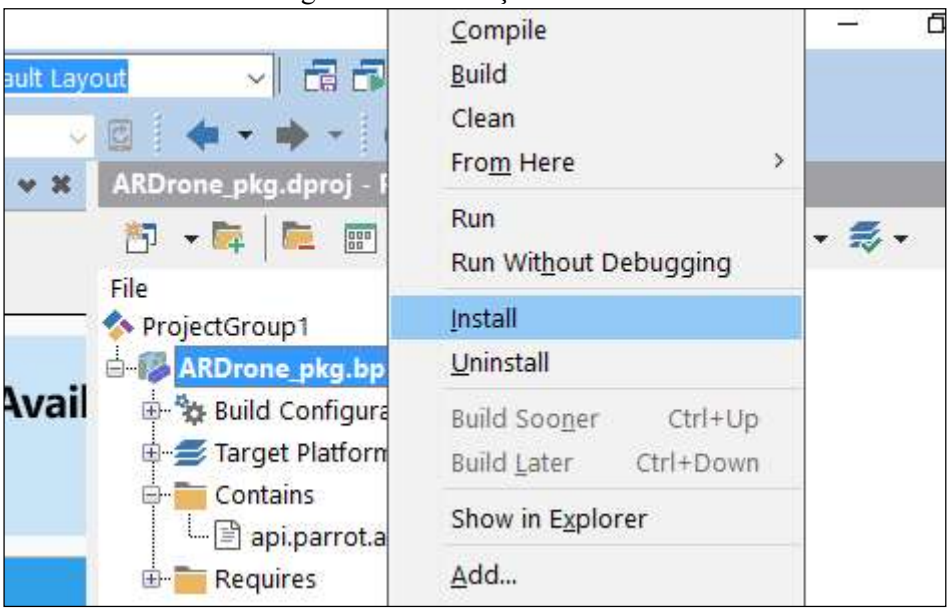
Para realização do desenvolvimento do AutoDrone utilizou-se um *notebook* Dell Inspiron 15 com sistema operacional Windows 10 64 bits, processador Intel Core I5 4210U

2,40 Giga Hertz (Ghz) e 8 Gigabytes (GB) de memória RAM. Para os testes foi utilizado o AR.Drone 2.0 Power Edition da Parrot.

3.3.1.1 Instalação do componente TArDrone

O código fonte do componente TArDrone pode ser obtido no repositório do GitHub do desenvolvedor Jim McKeeth[1] (MCKEETH, 2014). Após o *download* do código fonte é necessário abrir a IDE Delphi e abrir o arquivo do projeto do componente `ARDrone_pkg.dproj`. A próxima etapa é abrir janela `Project manager` do Delphi e nela localizar o arquivo ARDrone_pkg.bpl e clicar com o botão direito e selecionar a opção `Install`, conforme é demonstrado na Figura 17.

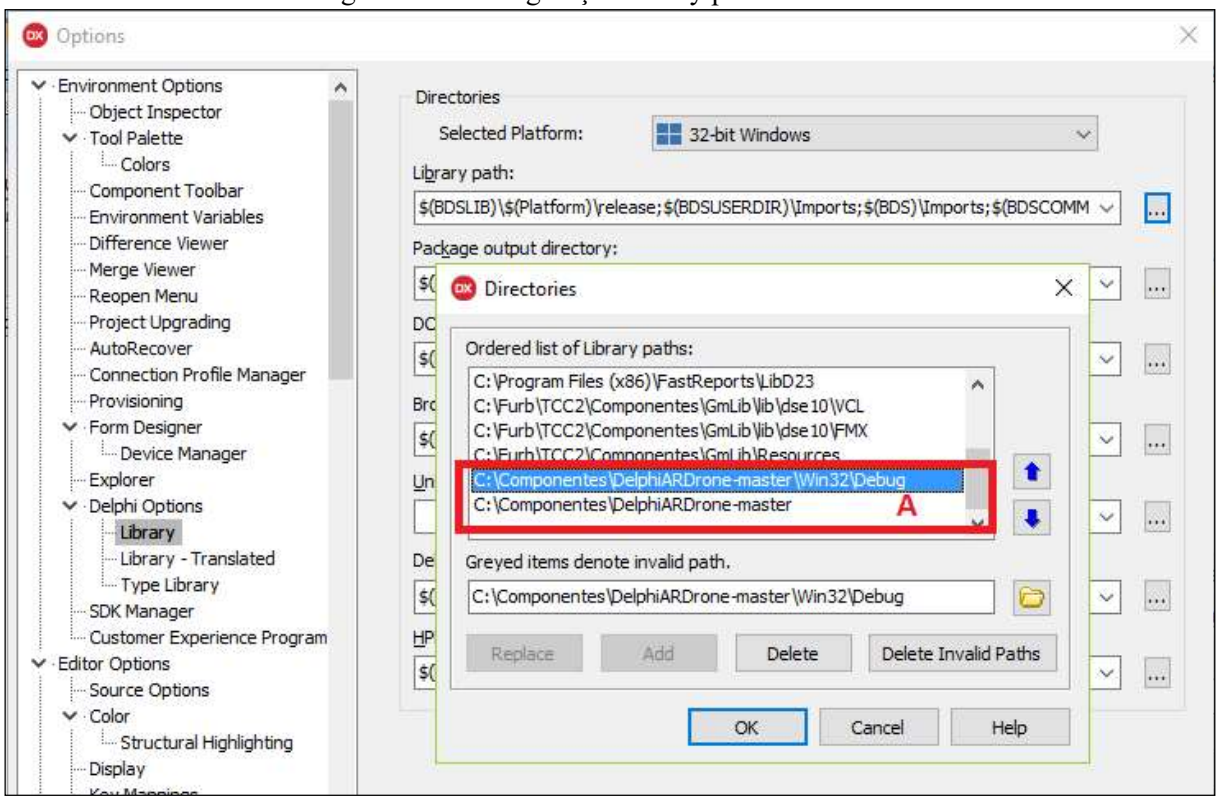
Figura 17 – Instalação `TArDrone`



Após a instalação do componente ainda é necessário configurar o `Library Path` do Delphi indicando os caminhos do código fonte do componente. O `library path` após a configuração é apresentado no item A em destaque na Figura 18.

[1] Disponível em: https://github.com/jimmckeeth/DelphiARDrone.

Figura 18 – Configuração library path `TArDrone`

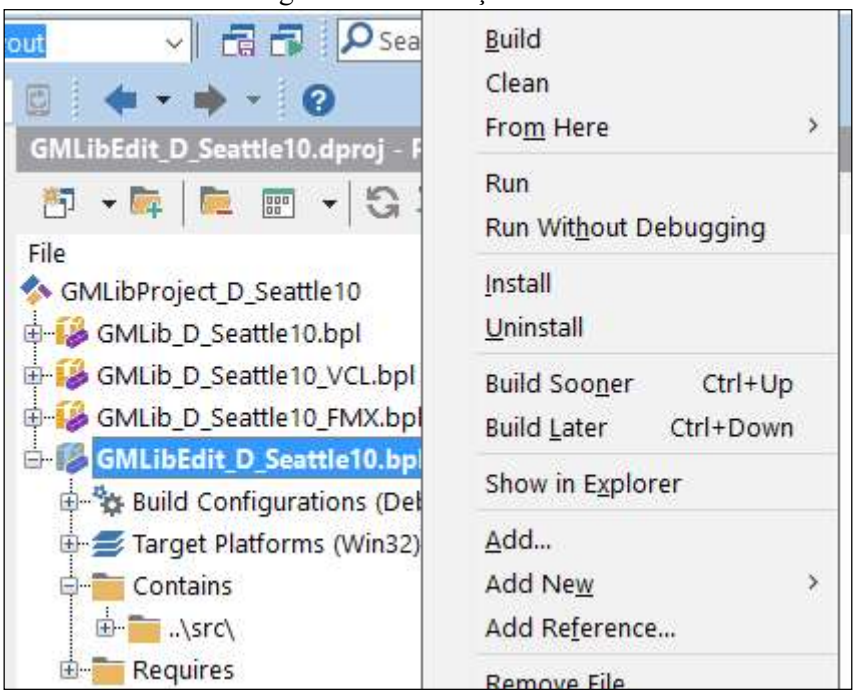


3.3.1.2 Instalação do pacote de componentes `GmLib`

O pacote de componentes `GmLib` pode ser obtido em seu repositório no SourceForge[2]. Após o *download* do arquivo `gmlib_1.5.3.zip`, é necessário descompactá-lo e abrir o arquivo do projeto correspondente a versão do Delphi, neste caso `GMLibProject_D_Seattle10.groupproj` para o Delphi 10 Seattle. A próxima etapa é abrir janela `Project manager` do Delphi e em todas as `bpls` é necessário clicar com o botão direito opção `Install`, conforme é demonstrado na Figura 19.

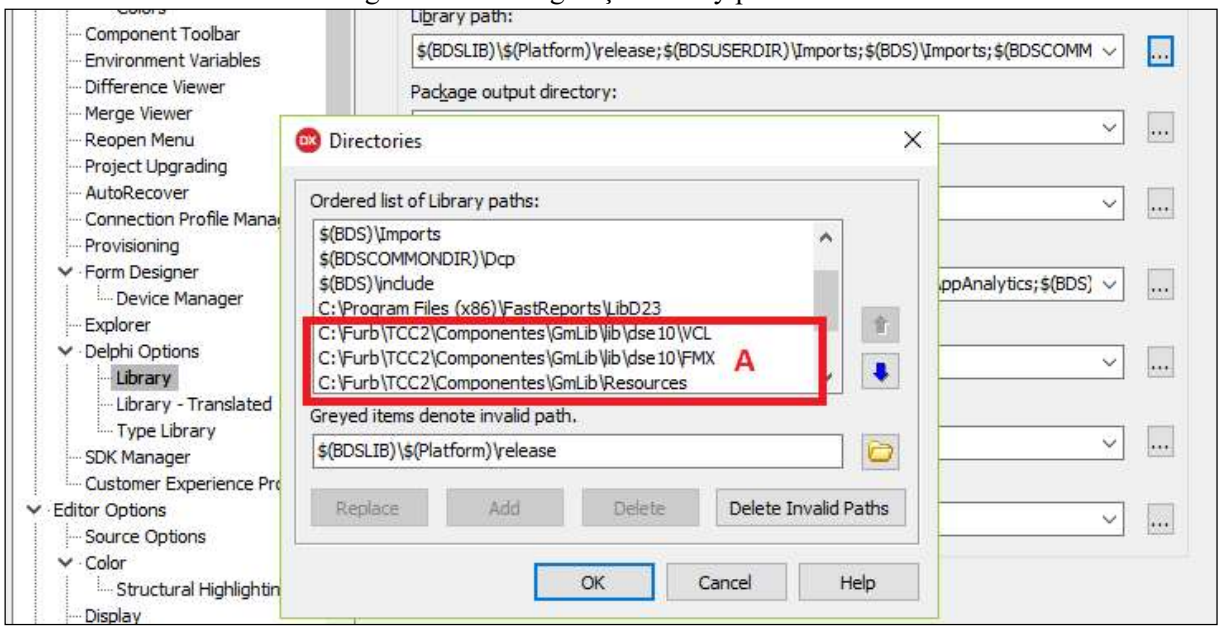
[2] Disponível em: https://sourceforge.net/projects/gmlibrary/?source=dlp.

Figura 19 – Instalação `GmLib`



Após a instalação é necessário configurar o `Library Path` do Delphi indicando os caminhos do código fonte do componente. O `library path` após a configuração é demonstrado no item A da Figura 20.

Figura 20 – Configuração library path `GmLib`



3.3.2 Arquivo de rotas

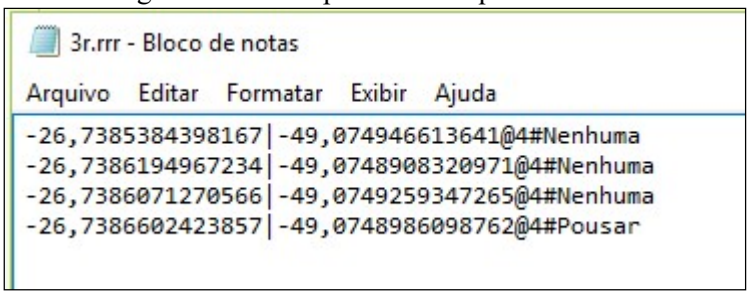
Quando utilizada a opção de. `Gravar rotas`, a aplicação gera dois arquivos no caminho configurado na guia configurações campo `path` rotas. O primeiro arquivo é uma imagem no

formato *bitmap*. Esta imagem é utilizada como auxílio visual no momento de selecionar uma rota pelo *grid* de gerenciamento de rotas.

O segundo arquivo armazena as informações das rotas. Este arquivo armazena o conteúdo em formato texto com a extensão `.rrr` onde cada linha representa um ponto. O formato texto foi escolhido pela sua facilidade de edição, desta maneira o usuário tem a opção de criar suas próprias rotas sem o uso da aplicação ou ainda de transportar suas rotas com facilidade sem a necessidade de algum programa de terceiros.

O formato das linhas de ponto pode ser observado na Figura 21 onde o arquivo está sendo editado no bloco de notas do sistema operacional Windows. O arquivo contém quatro linhas que correspondem a quatro pontos. Em cada linha tem-se como a primeira informação a latitude do ponto, este campo vai até o separador "|" (*pipe*). Em seguida temos a longitude que vai até o separador "@" e após isso temos a altura do ponto, seguido do separador "#" e por último a ação que o *drone* deve executar.

Figura 21 – Exemplo de um arquivo de rotas



Em não havendo ações a serem executadas em determinado ponto, deve ser informado o identificador `Nenhuma`.

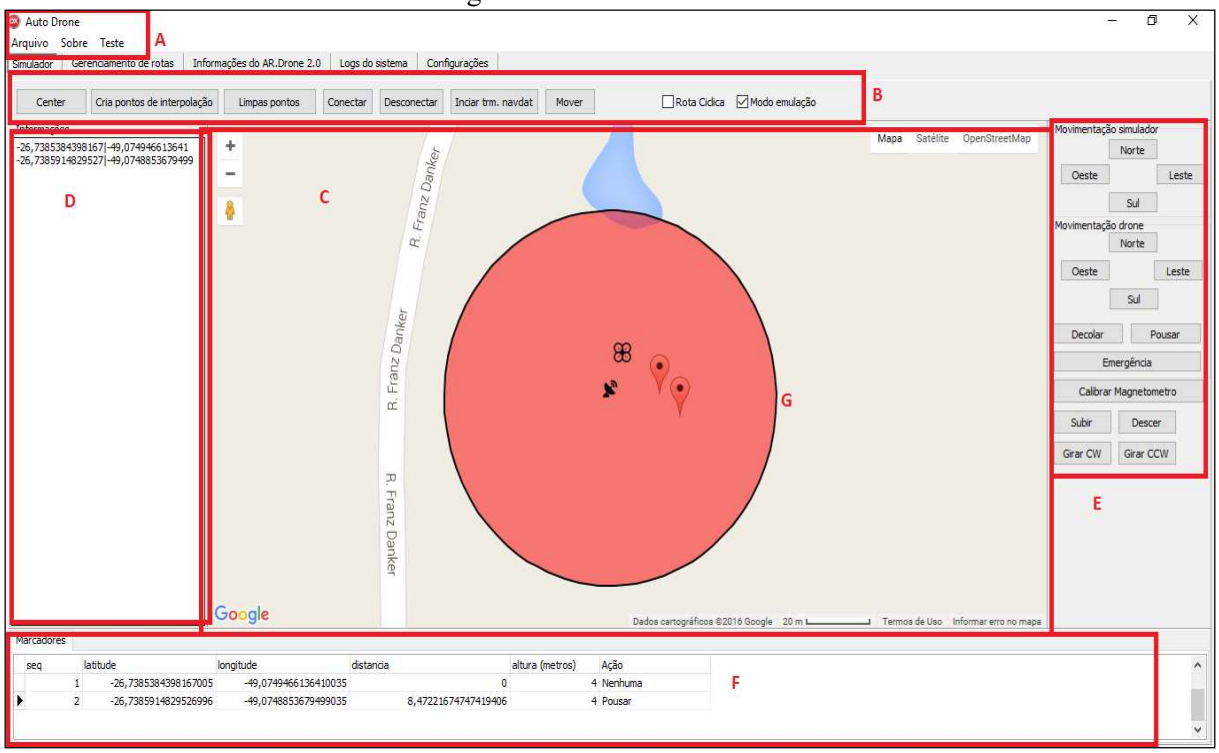
3.4 OPERACIONALIDADE DA IMPLEMENTAÇÃO

No AutoDrone, a principal atividade do usuário é interação com as rotas. Porém antes disso, são necessárias algumas etapas para sua criação, configuração e execução. Além destas, ainda existem outras funcionalidades secundárias para auxiliar nestas etapas. Esta seção apresenta de forma resumida a operacionalidade de cada uma das etapas supracitadas.

3.4.1 Guia Simulador

A tela principal da aplicação que pode ser vista na Figura 22 atem-se às funcionalidades referentes à criação das rotas, execução das rotas, conexão com o *drone* e controle do mesmo. Na Figura 22, no item `A` em destaque, encontra-se o menu principal com as opções `Arquivo, Sobre e Teste`.

Figura 22 – Guia Simulador



No menu `Arquivo` existem quatro funcionalidades, sendo elas: `Carregar arquivo` de rotas, recurso que permite carregar um arquivo com um itinerário de rotas previamente gravado pela aplicação; `Salvar rotas`, recurso que permite gravar um arquivo com todas as informações de pontos e ações configurados na aplicação; `Conectar mapa` que permite recarregar o mapa da aplicação, o que pode ser necessário se o aplicativo for aberto sem uma conexão com a internet e, finalmente, tem a opção de `Sair`, que encerra a aplicação.

Ainda no menu da aplicação existe a opção `Sobre`, que apresenta uma tela com informações da versão, data e autor da aplicação. Por último, o menu de teste que possui alguns comandos de controle de altitude para realizar testes com o *drone*.

Na Figura 22, no item em destaque B, se encontra uma série de botões com funcionalidades diversas. Da direita para esquerda o primeiro botão denominado `Center` posiciona o centro do mapa na coordenada configurada como sendo o ponto central. Este recurso é importante na situação em que é utilizada a aplicação corriqueiramente num mesmo ponto. Na sequência, está o botão `Criar pontos de interpolação`, o qual é responsável por analisar todas as subrotas e dividí-las de acordo com uma configuração de distância mínima entre pontos. Para o cálculo desta distância entre os pontos é utilizado a formula de Haversine (Figura 23), recurso que auxilia na precisão do caminhamento do *drone*.

Figura 23 – Implementação fórmula de Haversine

```
function TForm2.CalculaDistancia(Lat1, Lat2, Lng1, Lng2: Real): Real;
begin
  Result := 6371000*ARCCOS(COS(PI()*(90-Lat2)/180)*COS((90-Lat1)*PI()/180)+
  SIN((90-Lat2)*PI()/180)*SIN((90-Lat1)*PI()/180)*COS((Lng1-Lng2)*PI()/180));
end;
```

Em seguida temos o botão `Limpar pontos`, é responsável por limpar todos os pontos de rotas configuradas no mapa. Os próximos botões são `Conectar` e `Desconectar` respectivamente responsáveis por abrir e fechar o canal de comunicação com o *drone*.

O botão `Inciar trm Navdat` é responsável por iniciar a comunicação com o *drone* e dar início ao processo de recebimento das informações fornecidas pelo mesmo, informações essas necessárias que possibilitarão a aplicação conhecer o estado e a posição do *drone* em determinado momento. Por último, o botão `Mover`, que é responsável pelo início da movimentação do *drone* com base na rota configurada.

A caixa de seleção `Rota Cíclica`, quando marcada, oferece o recurso de gerar um ponto adicional no final da rota fazendo com que o *drone* retorne ao ponto de partida. E por último há a caixa de seleção `Modo de emulação`, a qual sendo selecionada altera o comportamento do botão `mover`. Neste caso, estando em modo de emulação apenas o *drone* do mapa é movimentado e não o *drone* real.

Na área central da Figura 22 no item em destaque C se encontra o mapa. Este mapa é fornecido pelo Google Maps e sendo assim ele possui recursos de *zoom* de visualização da área em 2D e visão por imagens captadas por satélite. Clicando-se no mapa será adicionado um ponto de referência. Esta é a forma utilizada pelo Autodrone para estabelecimento da rota que deverá ser utilizada pelo *drone*.

A Figura 22 no item em destaque G mostra o círculo vermelho que representa a área de cobertura do sinal de wireless do drone. Este recurso é meramente ilustrativo para informar ao usuário do limite do sinal tendo em vista evitar a criação de rotas fora do alcance do mesmo.

Na Figura 22 no item em destaque D, está um campo onde são adicionadas diversas informações da aplicação. Estas informações variam de acordo com o comando executado anteriormente. Como exemplo, pode-se citar mensagem de sucesso ou falha na gravação da rota, percentual disponível da bateria e pontos gerados pelo botão `Criar pontos de interpolação`.

Na Figura 22 no item em destaque E estão dispostos os controles de movimentação do simulador e do *drone* físico. Além disso, estão disponíveis comandos de pouso, decolagem, calibragem do magnetômetro, de giro no sentido horário e anti-horário do *drone*. Também está

disponível o botão de emergência, o qual quando ativado interrompe imediatamente o voo do *drone*.

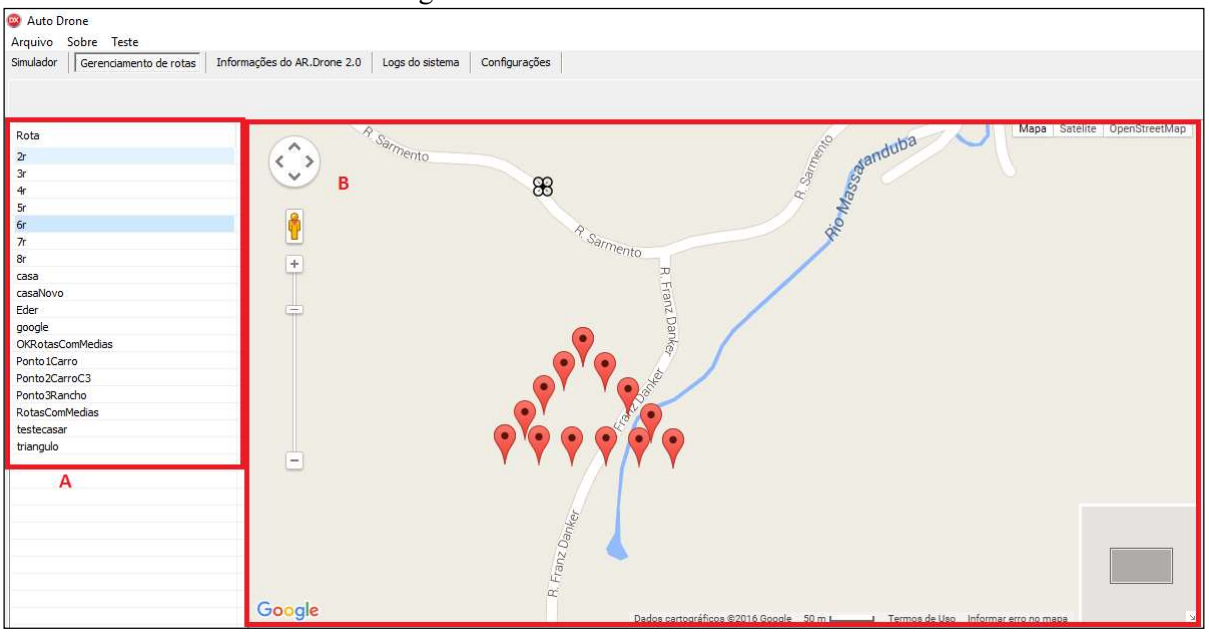
Na Figura 22 no item em destaque F está disponível um *grid* contendo a lista dos pontos no mapa. Neste *grid* também é possível adicionar pontos manualmente sem a ajuda do mapa. Além disso é demonstrada a distância em relação ao ponto anterior e é possível informar qual ação o *drone* irá executar quando atingir este ponto durante seu voo.

3.4.2 Guia Gerenciamento de rotas

A segunda guia da aplicação, denominada Gerenciamento de rotas, é dividida em duas áreas. A Figura 24 no item em destaque A mostra uma lista com todas as rotas gravadas no aplicativo dentro da pasta configurada na guia de configurações. Para que o arquivo com nome exibido na lista seja carregado, pode-se dar um duplo clique sobre ele ou clicar com o botão direito e selecionar a opção de `Carregar rota`. Ainda com um clique com o botão direito do mouse sobre um item pode-se excluir o arquivo.

Na Figura 24 no item em destaque B é possível observar uma imagem exibindo a rota referente ao item selecionado na lista da direita. Esta imagem foi capturada no momento em que a rota é gravada e também fica armazenada na pasta configurada.

Figura 24 – Gerenciamento de rotas

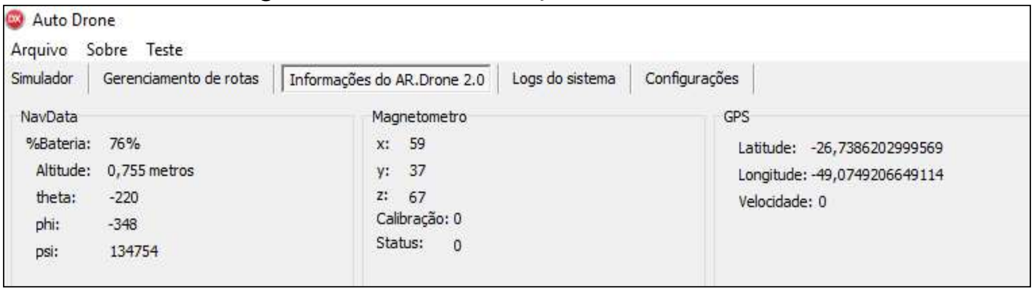


3.4.3 Guia Informações do AR.Drone 2.0

A Figura 25 mostra a guia de Informações do AR.Drone 2.0, que exibe uma série de informações fornecidas pelo mesmo. Entre as principais pode-se destacar as informações do

grupo NavData que contém as informações mais básicas do *drone*. A informações do GPS e magnetômetro.

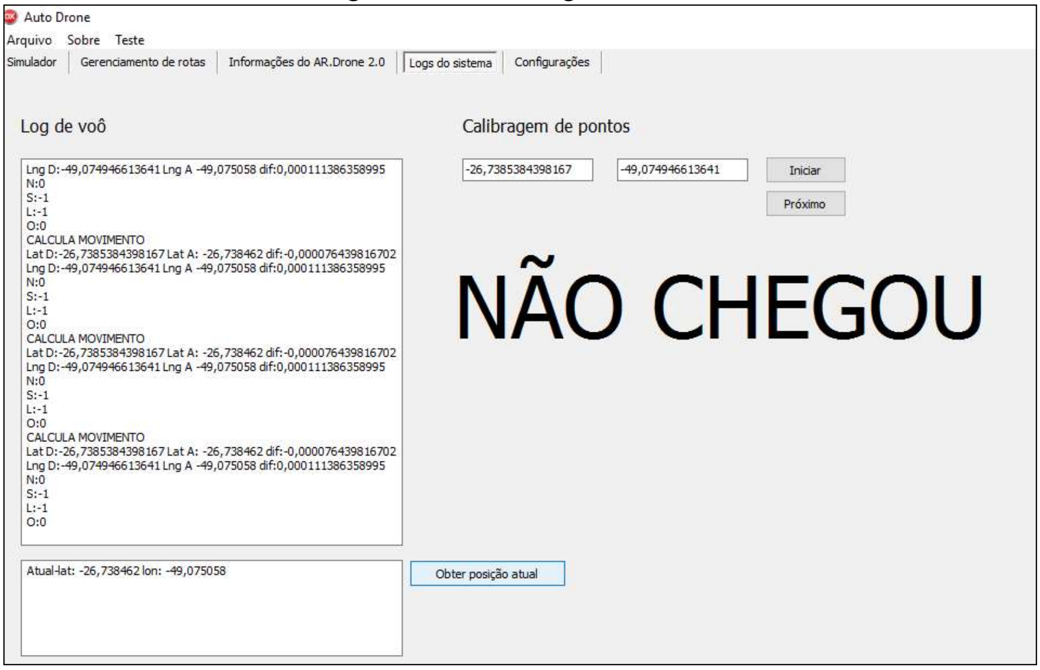
Figura 25 – Guia Informações do AR.Drone 2.0



3.4.4 Guia Logs do sistema

A Figura 26 mostra a guia de Logs do sistema a qual apresenta uma série de informações para o monitoramento.

Figura 26 – Guia Logs do sistema



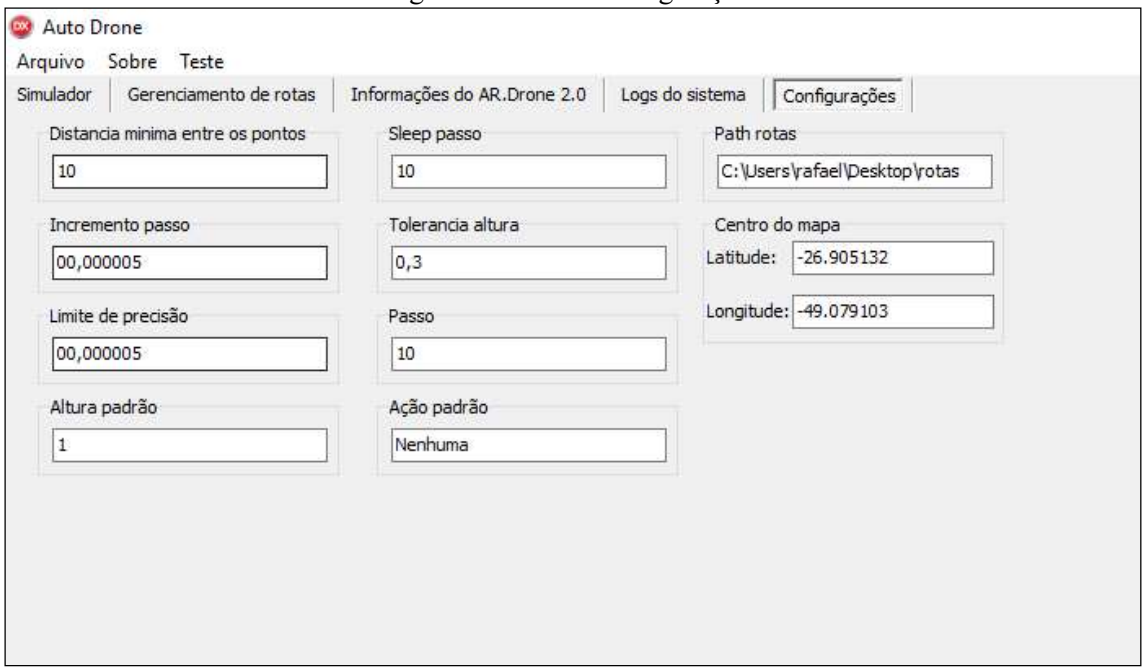
O campo do lado esquerdo consiste em um *log* de voo onde são mostradas todas as informações do cálculo executadas durando o voo bem como a localização em que o *drone* se encontra. Os demais campos da direita são ferramentas que permitem validar a precisão de um ponto da rota configurado.

3.5 GUIA CONFIGURAÇÕES

A aplicação possui uma série de configurações que tem o objetivo principal de calibrar a forma como os cálculos e as rotas são realizadas pelo drone. Conforme pode ser visto na Figura 27 as configurações disponíveis são:

a) `Distância mínima entre os pontos`: esta configuração é utilizada pela rotina de interpolação de pontos, esta informação determina a distância mínima em metros dos pontos que serão criados entre os já existentes;
b) `Incremento passo`: esta informação em graus decimais é utilizada no modo simulação, ela determina o tamanho do deslocamento do drone para determinada direção;
c) `Limite de precisão`: esta configuração em graus decimais é utilizada durante a movimentação do *drone* no momento que ele chegou ao ponto determinado pela rota, tendo como tolerância o valor informado;
d) `Altura padrão`: esta configuração determina a altura que será utilzada caso na configuração da rota não for informada nenhuma;
e) `Tempo espera pós-passo`: esta configuração determina um tempo em milisegundo utilizada na movimentação do drone com o objetivo de obter uma posição de latitude e longitude atual com mais precisão;
f) `Tolerância altura`: determina a tolerância em centímetros para quando houver um comando de alteração de altura;
g) `Passo`: configuração que determina o tamanho do passo que o *drone* irá efetuar, esta informação é repassada para o componente TArDrone;
h) `Ação padrão`: esta configuração determina a ação por padrão que será executada quando o *drone* chegar no ponto configurado e não houver nenhuma ação informada;
i) `Path rotas`: determina a partir de qual diretório será carregado automaticamente às rotas disponíveis na guia de gerenciamentos de rotas e disponíveis na grid;
j) `Centro do mapa`: é uma latitude e longitude que será utilizada pelo botão de `center` da guia `simulador` para posicionar como centro do mapa.

Figura 27 – Guia Configurações



3.6 OBTENÇÃO DOS DADOS

Para se obter os dados do Ar.Drone 2.0 é necessário estabelecer-se um canal de comunicação via UDP na porta 5556 para o envio de comandos para o *drone*, funcionalidade esta que o componente `TArDrone` já fornece. Para o recebimento das informações deve ser estabelecido um outro canal de comunicação UDP na porta 5554.

Quadro 16 – Configuração do retorno

```
1   Var
2     Buf: TIdBytes;
3   Begin
4     IdUDPServer1.Active := True;
5     Buf[0] := 1;
6     ARDrone.NavUDP.SendBuffer(buf);
7     ARDrone.Config('general:navdata_demo','TRUE');
8     ARDrone.Config('general:navdata_options','22');
9     ARDrone.Config('general:navdata_options','777060865');
10    ARDrone.SendCommand('AT*CTRL','0');
```

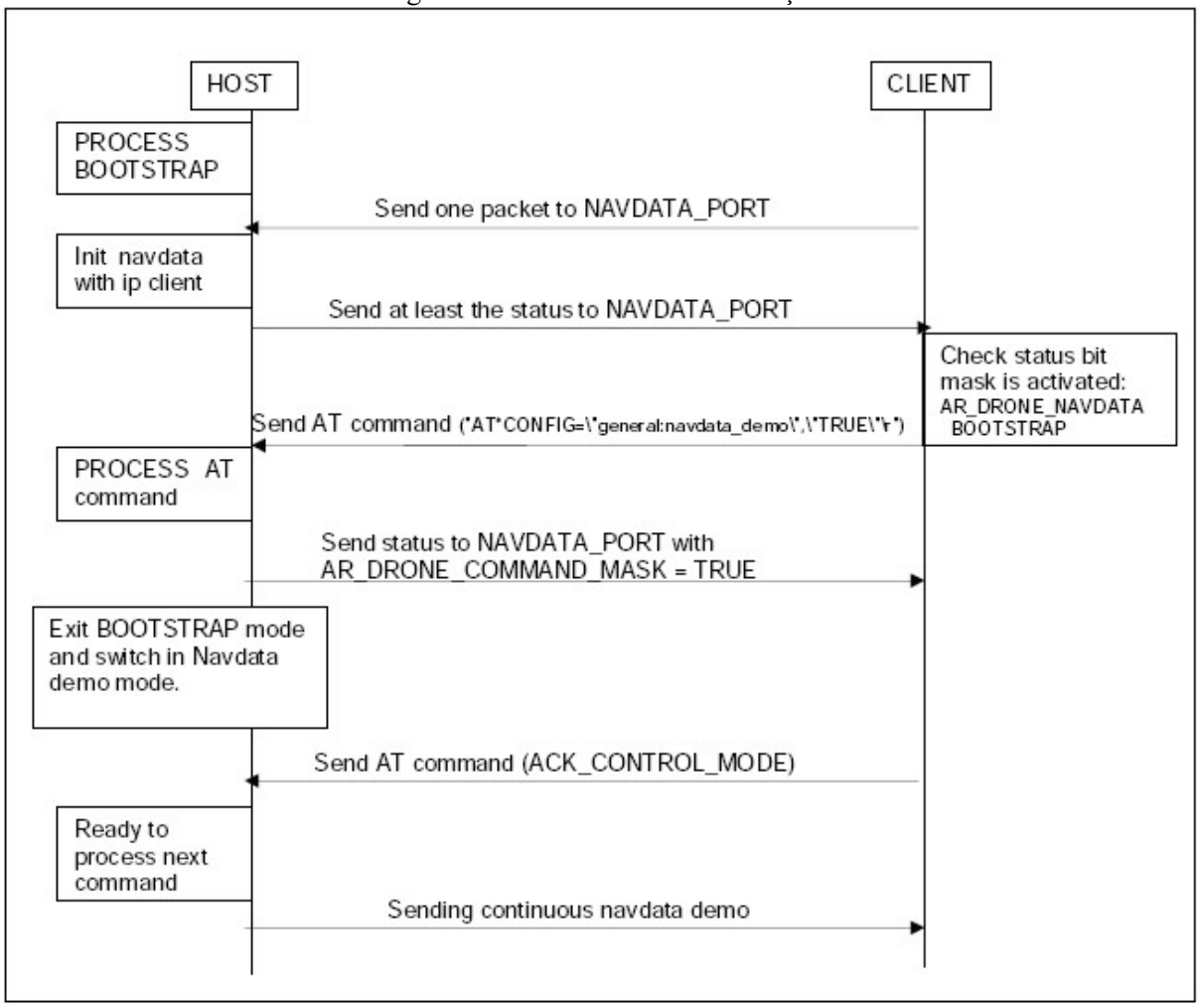
Após o estabelecimento da conexão, é necessário enviar o comando de configuração para o estabelecimento de quais informações o *drone* irá enviar. Para efetuar esta configuração deve-se utilizar o comando `general:navdata_demo` para configurar o Ar.Drone 2.0 sair do

modo *bootstrap* e assim o *drone* será inicializado de forma que possa enviar seus dados (Quadro 16, linha 9).

Em seguida deve-se enviar o comando `general:navdata_options` com o ID das *tags* que deseja receber. E por último deve-se enviar um comando ACK para que o *drone* comece a enviar os dados das *tags* solicitadas, como pode ser visto no Quadro 16 nas linhas 9 a 13.

Uma vez realizada a configuração e estabelecida a comunicação, é necessário enviar continuamente um sinal de comunicação para que a conexão não seja fechada, este sinal deve ser enviado a um intervalo inferior a 50 milisegundos. Caso não houver nenhum tráfico de dados neste intervalo o *drone* irá enviar uma resposta `ARDRONE_COM_WATCHDOG_MASK` e saíra do modo *bootstrap* e por consequência ele para de enviar os dados. Caso isso ocorra pode se enviar um comando `AT*COMWDG` para reestabelecer a comunicação. Este processo é demonstrado no diagrama de sequência da Figura 28.

Figura 28 – Processo de comunicação



Se o tempo entre os comandos alcançar 2000 ms o *drone* enviará um comando de aviso `ARDRONE_COM_LOST_MASK` e irá interromper toda a comunicação. E neste caso a comunicação

só poderá ser retomada reinicializando a conexão. O Quadro 17 mostra essa comunicação sendo efetuada por uma *Thread*. Na linha 9 é enviado o sinal de `ACK` e na linha 12 é obtida a resposta do *drone.* Na linha 13 esta resposta é armazenada em uma lista para ser interpretada posteriormente.

Quadro 17 – Recebimento das informações

```
1   procedure TForm2.IdThreadDadosRun(Sender: TIdThreadComponent);
2   Var
3     buf: TIdBytes;
4   begin
5     if FEnviaNovaSolicitacao then
6     begin
7       ARDrone.NavUDP.Send('1');
8       SetLength(Buf, 2048);
9       ARDrone.navUdp.ReceiveBuffer(buf);
10      bufferDados.Add(buf);
11    end;
12  end;
```

3.6.1 Interpretação dos dados obtidos

Após o recebimento dos dados (o que ocorre em um array de bytes), os mesmos devem ser interpretados. Para isto os dados são fracionados em diferentes estruturas. O *layout* básico de uma estrutura é apresentado no Quadro 18.

Quadro 18 – Layout da estrutura

Header	**Drone**	**Sequence**	**Vision**	**Option 1**			...	**Checksum block**		
0x55667788	**state**	**number**	**flag**	id	size	data	...	cks id	size	cks data
32-bit	32-bit	32-bit	32-bit	16-bit	16-bit	...	...	16-bit	16-bit	32-bit
int.	int.	int.	int.	int.	int.	...	...	int.	int.	int.

O formato básico das estruturas enviadas pelo drone são obrigatoriamente formadas por um *header* que identifica o tipo da estrutura, em seguida o campo `drone state` onde é informado o status atual do *drone*, seguido na sequência do comando enviado e *`vision flag`*. Após estas informações vem um conjunto de dados referente a cada estrutura. Finalmente e tendo por objetivo garantir a integridade dos dados é enviado o *checksum* do pacote. As *tags* fornecidas pelo *drone* e suas estruturas são detalhadas no apêndice A.

As *tags* utilizadas pela aplicação são: `NAVDATA_DEMO_TAG` que fornece as informações básicas do *drone*, a `NAVDATA_MAGNETO_TAG` que fornece os dados referentes à bússola virtual e `NAVDATA_GPS_TAG` que fornece os dados da leitura do GPS. As Figuras 29, 30 e 31 apresentam

estas estruturas completamente mapeadas, tendo como referência a aplicação de exemplo fornecida pelo fabricante na linguagem C++.

Figura 29 – Estrutura da NavDat_Demo

```
NAVDATA_DEMO = record
    tag : Word;
    size : Word;
    ctrl_state: cardinal;
    vbat_flying_percentage: cardinal;
    theta: Single;
    phi: Single;
    psi: Single;
    altitude: Integer;
    vx: Integer;
    vy: Integer;
    vz: Integer;
    num_frames: cardinal;                 // Don't use
    detection_camera_rot: vector31_t;       // Don't use
    detection_camera_trans: vector31_t;     // Don't use
    detection_tag_index: cardinal;        // Don't use
    detection_camera_type: cardinal;      // Don't use
    drone_camera_rot: vector31_t;           // Don't use
    drone_camera_trans: vector31_t;         // Don't use
end;
```

Figura 30 – Estrutura da NavDat_Magneto

```
// Magneto
NAVDATA_MAGNETO = record //coordenadas polares
    tag: Word;
    size: Word;
    mx: smallint;
    my: smallint;
    mz: smallint;
    magneto_raw: vector31_t;// magneto in the body frame, in mG
    magneto_rectified: vector31_t;
    magneto_offset: vector31_t;
    heading_unwrapped: single;
    heading_gyro_unwrapped: single;
    heading_fusion_unwrapped: single;
    magneto_calibration_ok: byte;
    magneto_state: cardinal;
    magneto_radius: single;
    error_mean: single;
    error_var: single;
    tmp1: single;
    tmp2: single;
end;// magneto;
```

Figura 31 – Estrutura da NavDat_GPS

```
NAVDATA_GPS = record
      tag: Word;
      size: Word;
      lat: double;                  //*!< Latitude */
      lon: double;                  //*!< Longitude */
      double: double;             //*!< Elevation */
      hdop: double;                 //*!< hdop */
      data_available : cardinal;       //*!< When there is data available */
      unk_0: array [0..7] of byte;
      lat0:double;                 //*!< Latitude ??? */
      lon0:double;                 //*!< Longitude ??? */
      lat_fuse:double;             //*!< Latitude fused */
      lon_fuse:double;             //*!< Longitude fused */
      gps_state: cardinal;            //*!< State of the GPS, still need to figure out */
      unk_1:  array [0..39] of byte;
      vdop:double;                 //*!< vdop */
      pdop:double;                 //*!< pdop */
      speed: Single;                //*!< speed */
      last_frame_timestamp: cardinal; //*!< Timestamp from the last frame */
      degree: Single;               //*!< Degree */
      degree_mag: Single;           //*!< Degree of the magnetic */
      unk_2: array [0..15] of byte;
      channels: array [0..11] of TChannels;
      gps_plugged: integer;         //*!< When the gps is plugged */
      unk_3: array [0..107] of byte;
      gps_time: double;            //*!< The gps time of week */
      week  : Word;                //*!< The gps week */
      gps_fix: byte;              //*!< The gps fix */
      num_sattelites: byte;       //*!< Number of sattelites */
      unk_4:  array [0..23] of byte;
      ned_vel_c0:double;          //*!< NED velocity */
      ned_vel_c1:double;          //*!< NED velocity */
      ned_vel_c2:double;          //*!< NED velocity */
      pos_accur_c0:double;        //*!< Position accuracy */
      pos_accur_c1:double;        //*!< Position accuracy */
      pos_accur_c2:double;        //*!< Position accuracy */
      speed_acur: Single;          //*!< Speed accuracy */
      time_acur: Single;           //*!< Time accuracy */
      unk_5: array [0..71] of byte;
      temprature: Single;
      pressure: Single;
end;
```

Conhecendo-se a estrutura das informações enviadas pelo drone, o próximo passo é separá-las de modo a viabilizar a utilização das mesmas pelo aplicativo. O Quadro 19 apresenta o trecho de código onde é realizado este processo. Como a quantidade de *tags* de retorno não é constante a leitura é feita tag por *tag* dentro do laço *while* da linha 10. Inicialmente é copiado uma parte do *buffer* para identificar qual é o tipo de *tag* que está sendo lida (linhas 13 e 14).

Em seguida é realizada uma nova leitura utilizando o comando `copyBuf` e `move` para copiar um pedaço da memória para uma nova posição visto que o destino é uma estrutura mapeada conforme foi demonstrado no Quadro 19. Após a cópia os valores já estarão devidamente separados dentro das estruturas declaradas. Este processo pode ser visto na linha 28 e 29 onde são copiados os dados para a variável `ardrone_navdata_demo`.

Nas linhas 39 e 40 são realizadas as cópias dos dados do magnetômetro para a variável `ardrone_navdata_magneto` e nas linhas 58 e 59 são realizadas as cópias dos dados do GPS

para a variável `ardrone_navdata_gps`. Após este processo pode-se fazer uso das variáveis para a leitura dos dados.

Quadro 19 – Quebrando o retorno

```
      Var
        ardrone_navdata_demo: navdata_demo;
        ardrone_navdata_magneto: navdata_magneto;
        ardrone_navdata_gps: navdata_gps;
      begin
      index := 16;
        loop := false;
        while (index < TAMANHO ) do
        begin
          CopyBuf(buf, dados_mov, index
          move(dados_mov, tmp_tag, 2);
          index := index + 2;
          CopyBuf(buf, dados_mov, index);
          move(dados_mov, tmp_size, 2);
          index := index + 2;
          index := index - 4;
          case tmp_tag of
              ARDRONE_NAVDATA_DEMO_TAG :
              begin
                CopyBuf(buf, dados_mov, index);
                move(dados_mov, ardrone_navdata_demo, MIN(tmp_size,
sizeof(NAVDATA_DEMO)));
                if loop then
                  Break;
                loop := true;
              end;
              ARDRONE_NAVDATA_MAGNETO_TAG:
              begin
                CopyBuf(buf, dados_mov, index);
                move(dados_mov, ardrone_navdata_magneto, MIN(tmp_size,
sizeof(NAVDATA_MAGNETO)));
              end;
              ARDRONE_NAVDATA_GPS_TAG:
              begin
                CopyBuf(buf, dados_mov, index);
                move(dados_mov, ardrone_navdata_gps, MIN(tmp_size,
sizeof(NAVDATA_GPS)));
               break;
              end;
```

3.6.2 Cálculo da direção

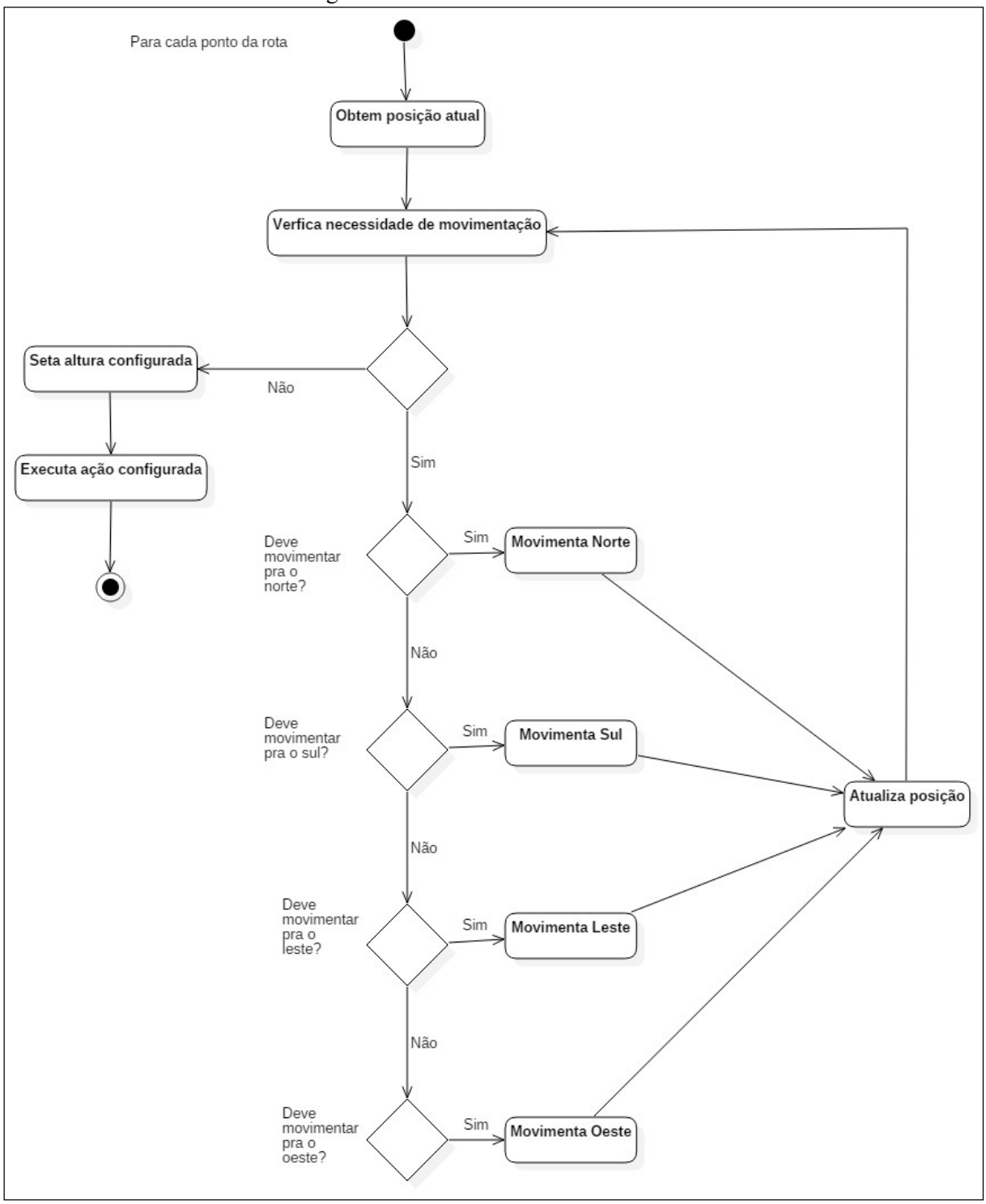
Após os dados lidos e interpretados (Quadro 19), a aplicação realiza o cálculo para determinar a direção que *drone* de seguir. Este processo ocorre durante todo o voo do *drone*, e para tanto é utilizado como referência sua posição atual e a posição do próximo ponto da rota.

O Quadro 20 apresenta o cálculo da direção que o drone deve seguir, o cálculo é realizado considerando a posição atual e o ponto objetivo da rota. Após o cálculo é retornado se o drone deve ser movimento e em quais direções.

Quadro 20 - Cálculo da direção

```
1  function TForm2.DeveMovimentar(var Norte, Sul, Leste, Oeste: Boolean;
2  Lat1, Lat2, Lng1, Lng2: Real): Boolean;
3  begin
4    Norte := ((Lat2-Lat1) > 0) and ((Lat2 - Lat1) > LIMITE_PRECISAO);
5    Sul := ((Lat2-Lat1) < 0) and ((Lat2 - Lat1) < LIMITE_PRECISAO * -1);
6    Leste := ((Lng2-Lng1) > 0) and ((Lng2-Lng1) > LIMITE_PRECISAO);
7    Oeste := ((Lng2-Lng1) < 0) and ((Lng2-Lng1) < LIMITE_PRECISAO * -1);
8    Result := Norte or Sul or Oeste or Leste;
9  end;
```

Figura 32 - Processo de caminhamento



O diagrama atividades da Figura 32 apresenta o macro processo que determina o cálculo da movimentação. Inicialmente é obtida a posição atual, em seguida é realizado um cálculo com a latitude e longitude do *drone* com o próximo ponto da rota. Após este cálculo é possível identificar para qual posição cardeal o *drone* deve se movimentar. Chegando no ponto estabelecido é setada a altura do drone e executada a ação configurada na rota. Por último, após cada passo o *drone* atualiza sua posição atual e ciclo recomeça.

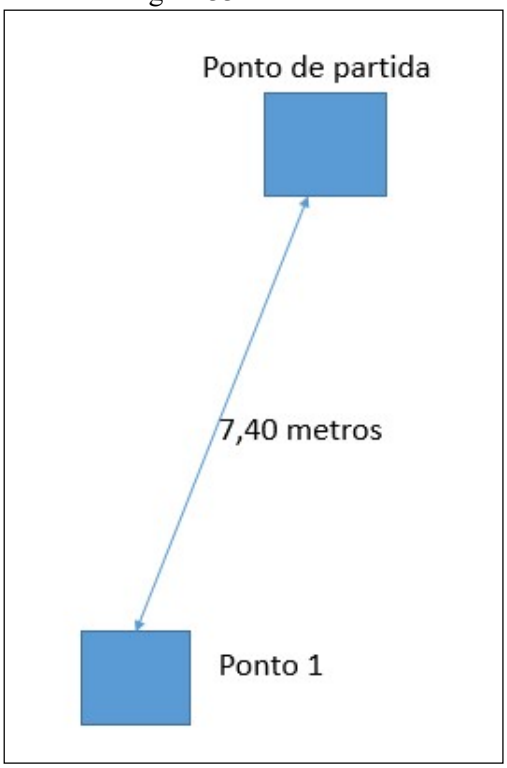
3.7 RESULTADOS E DISCUSSÕES

Os resultados obtidos foram validados através de uma série de testes de validação. O objetivo principal foi aferir a posição alcançada pelo *drone* em relação a posição dos pontos definidos na rota. Para realizar estas medições foram criados 3 cenários, cada um com características e finalidades distintas.

3.7.1 Cenário 1

A rota programada para o *drone* executar é de apenas um ponto a uma distância de sete metros e quarenta centímetros do ponto de origem (Figura 33). Ao chegar a este ponto ele deve pousar. Após a execução da missão dez vezes se obteve uma média de 1,51 metros de precisão em relação ao alvo.

Figura 33 – Cenário 1



As distâncias encontradas em cada tentativa podem ser visualizadas na Tabela 1. Foram observados dois aspectos que merecem destaque: (a) a interferência de vento prejudica o resultado e, (b) se a carga da bateria estiver com menos de vinte e cinto por cento a precisão da rota também é prejudicada, pois a oscilação dos dados do GPS é maior.

Tabela 1 – Resultados cenário 1

Tentativa	Distância do ponto objetivo em metros
1	2,7
2	1,4
3	1,8
4	1,5
5	0,8
6	1,6
7	0,5
8	2,3
9	1,6
10	0,9
Média	1,51

3.7.2 Cenário 2

Para o segundo cenário de testes foi utilizado uma rota com três pontos (Figura 34). O ponto um ficou a uma distância de 7,40 metros do ponto de partida, o ponto dois ficou a uma distância de 5,30 metros do ponto um e o ponto três ficou a 8,10 metros do ponto dois. A programação deste cenário estabelece que o *drone* irá pousar ao atingir o ponto três.

A Tabela 2 apresenta os dados coletados onde fica caracterizado que foi obtido um índice de precisão de 1,63 metros em média, após as dez execuções do roteiro. O objetivo deste teste foi o de identificar o nível de precisão em rotas com mais de um ponto.

Figura 34 – Cenários 2 e 3

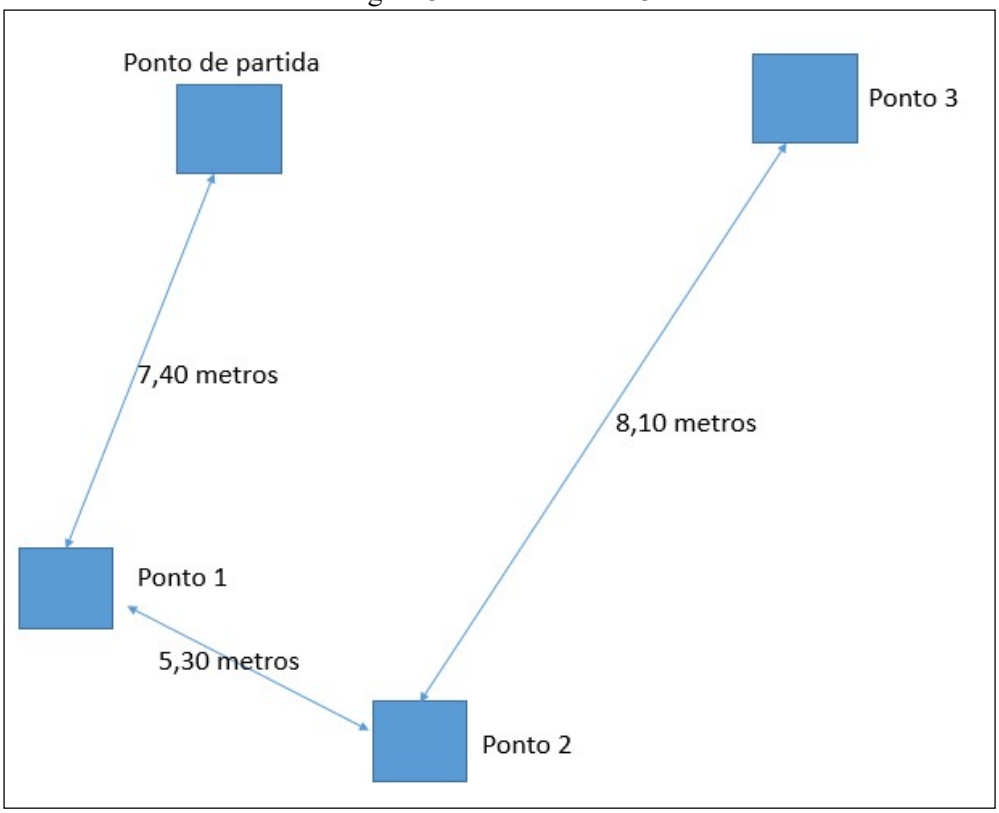


Tabela 2 – Resultados cenário 2

Tentativa	Distância do ponto final objetivo e metros
1	1,8
2	1,2
3	1,3
4	1,8
5	0,7
6	1,2
7	0,9
8	3,5
9	2,1
10	1,8
Média	1,63

3.7.3 Cenário 3

No cenário três foram utilizados os mesmos pontos do cenário dois (Figura 34), no entanto foram adicionadas alterações de altura nos pontos um e dois. No momento da partida a altura do *drone* era de 80 centímetros. Ao chegar no ponto 1 o *drone* alterou sua altura para 1,5 metros. No ponto 2 a altura foi alterada para 2,00 metros e chegando ao ponto 3 o *drone* pousou.

Como pode ser visto na Tabela 3 houve pouca alteração em relação a precisão média para o cenário anterior. Foi atingida uma média de precisão de 1,65 metros em relação ao último ponto.

Tabela 3 – Resultados cenário 3

Tentativa	Distância do ponto final objetivo em metros
1	1,2
2	1,5
3	1,4
4	1,6
5	1,1
6	1,9
7	1,5
8	1,9
9	2,1
10	2,3
Média	1,65

3.7.4 Considerações

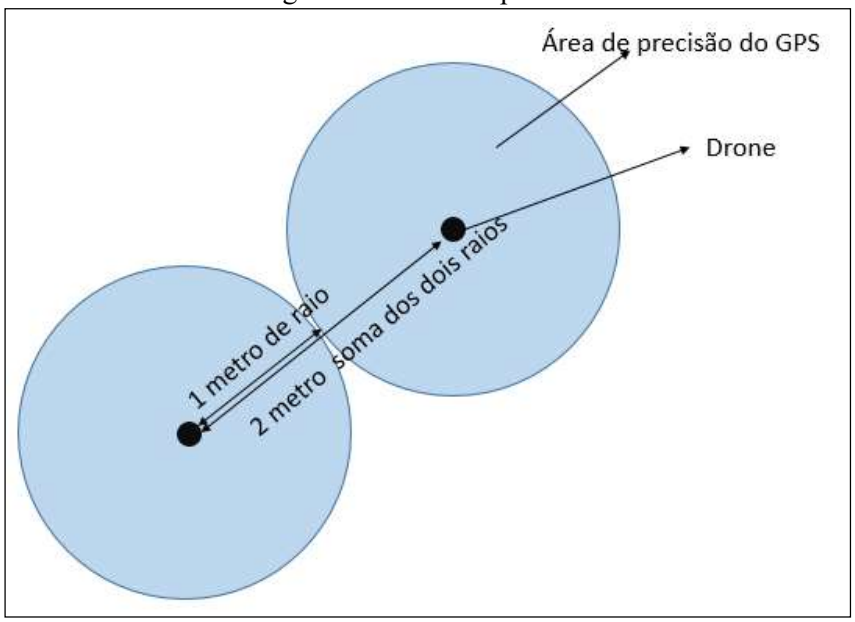
Ao longo do desenvolvimento do trabalho e particularmente durante a fase de execução dos testes, percebeu-se algumas dificuldades na utilização do *drone*. Um problema que ocorre esporadicamente é que o *drone* para de responder. A busca por uma solução para este problema não resultou em sucesso. Isto dificultou a fase de testes pois a cada ocorrência do problema a etapa do cenário de testes precisava ser reiniciada. A suposição levantada é que este problema possa estar no sequenciamento dos pacotes de comunicação trocados entre o *drone* e o computador e/ou por perda de pacotes.

Outra dificuldade encontrada foi o mapeamento dos pontos com exatidão, pois a leitura do GPS do *drone* tem pelo menos um metro de raio de precisão (conforme as informações do fabricante). A Tabela 4 mostra uma série de leituras efetuadas pelo GPS do *drone* em uma mesma posição onde é possível ver a variação dos dados recebidos. Este efeito de oscilação é um dos fatores de ruído que tornam mais complexa a tarefa de fazer o *drone* seguir uma rota pré-programada. No pior cenário obteve-se um erro de até dois metros em relação ao ponto de referencia considerado. A situação é demonstrada na Figura 35.

Tabela 4 – Leitura do GPS

Leitura	Latitude	Longitude
1	-26,7386032561209	-49,0749366406442
2	-26,7386032447869	-49,0749366123516
3	-26,7386032447869	-49,0749366123516
4	-26,7386032590294	-49,0749366208919
5	-26,738603266921	-49,0749366275052
6	-26,7386032712725	-49,074936635009
7	-26,7386032712725	-49,074936635009
8	-26,7386032459094	-49,0749366131021
9	-26,7386045234764	-49,0749377702485
10	-26,7386045442017	-49,0749377003143

Figura 35 – Área de precisão



Algumas alternativas foram consideradas na tentativa de minimizar o erro percebido aplicando-se, por exemplo, o cálculo de média e desvio padrão dos dados coletados em uma determinada janela de tempo. Contudo, nenhuma das alternativas consideradas mostrou-se eficiente. Como a janela de tempo para a recepção de um novo comando é de cinquenta milissegundos, cálculos mais complexos não foram considerados.

4 CONCLUSÕES

O objetivo principal deste trabalho foi o desenvolvimento de um aplicativo que permite automatizar rotas em um *drone* modelo AR.Drone 2.0 sem a necessidade de conhecimento específico de programação. Para isto foi criada interface gráfica onde é possível estabelecer-se uma rota e encaminhar esta rota para o *drone* para que o mesmo possa executá-la. Este objetivo foi atendido.

Durante o processo de desenvolvimento e testes, foram identificadas limitações técnicas as quais dificultaram a obtenção de resultados mais precisos. A aplicação desenvolvida implementa todo o conjunto de requisitos funcionais propostos além de implementar recursos para minimizar os erros de deslocamento quando utilizadas rotas muito extensas. Além disso, foi desenvolvido um formato de arquivo onde pode ser importado contendo rotas para o aplicativo de forma a possibilitar que um roteiro possa ser estabelecido utilizando-se editores de texto simples como o bloco de notas do Windows.

Em relação aos objetivos do trabalho, podemos considerar que o software desenvolvido para suporte ao estabelecimento de rotas, foi plenamente atendido visto que a aplicação possui todos os recursos necessários para criação e manutenção de rotas para o AR.Drone 2.0. Também foi possível criar um conjunto de cenários e validar as rotas e os algoritmos de condução do *drone* pela rota.

A ferramenta de desenvolvimento escolhida foi o ambiente Delphi 10 Seattle o qual se mostrou robusto e facilitou o desenvolvimento da aplicação. O componente TArDrone também atendeu as expectativas em relação ao que se propôs a fazer.

Como principais contribuições deste trabalho podem-se considerar:

a) disponibilizar uma aplicação que permita o estabelecimento de uma rota pre-programada para o drone executar;
b) documentação acerca do formato para realizar-se a leitura dos dados do *drone*, modo de interpretação bem como sua utilização e o mapeamento de toda as respostas do AR.Drone 2.0 (as quais até o momento não estavam disponíveis utilizando Delphi na literatura consultada).

4.1 EXTENSÕES

Como sugestões de futuras extensões ao trabalho estão:

a) implementar mais ações para o *drone* executar (tirar uma foto, abrir comunicação de vídeo, girar, realizar uma animação, retornar);
b) pesquisar e desenvolver um cálculo melhor para otimizar a precisão, uma alternativa

é também fazer o uso dos acelerômetros;

c) implementar recursos para utilização das câmeras;
d) automatizar para que o *drone* se alinhe com o norte sem a necessidade do usuário fazer isto;
e) extender o componente TAr.Drone implementando também as funcionalidades para recebimento das informações do *drone;*
f) criação de uma versão para *tablets*.

REFERÊNCIAS

AFONSO, M. P. Johnnatan. **Framework para sistema de navegação de veículos não tripulados**. 2014. Trabalho de Graduação (Ciência da Computação) – Instituto de Ciências Exatas e Biológicas, Universidade Federal de Ouro Preto, Ouro Preto, Brasil.

ALBUQUERQUE, Caio. **Uso de drones na agricultura de precisão é tema de estudo na Esalq**. 2013. Disponível em: < http://www5.usp.br/31918/uso-de-drones-na-agricultura-de-precisao-e-tema-de-estudo-na-esalq/>. Acesso em: 28 mar 2015.

AMARAL, Nelson de Frias. **Geolocalização em atividades físicas: Aplicação e expectativas.** 2014. 154 f. Dissertação (Mestrado) - Curso de Engenharia Informática, Instituto Superior de Engenharia do Porto, Porto, 2014. Disponível em: <http://recipp.ipp.pt/handle/10400.22/6229>. Acesso em: 21 maio 2016.

D'ALGE, J. C. L. **Cartografia para Geoprocessamento**. 2001- Instituto Nacional de Pesquisas Espaciais, São José dos Campos, Brasil. Disponível em: <http://mtc-m12.sid.inpe.br/col/sid.inpe.br/sergio/2004/04.19.14.47/doc/cap6-cartografia.pdf>. Acesso em: 21 maio 2016.

DEVELOPER. **Firemonkey component to control Parrot AR.Drone 2.0 Quadricopter.** 2014. Disponível em: <http://firemonkeytutorial.com/delphi-firemonkey-component-control-parrot-ar-drone-2-0-quadricopter/>. Acesso em: 20 fev 2016.

FALCONI, Carlos Eduardo. **Navegação Aérea – Como converter Coordenadas Geográficas.** 2009. Disponível em: <http://www.pilotopolicial.com.br/navegacao-aerea-como-converter-coordenadas-geograficas/>. Acesso em: 18 abr. 2016.

NASCIMENTO, A. David. **Análise de requisitos de hardware em um projeto UAV quadrotor**. 2011. Trabalho de Conclusão de Curso (Bacharel em Engenharia de Computação) - Escola Politécnica de Pernambuco – Universidade de Pernambuco.Pernambuco, Brasil.

MCKEETH, J. **Connecting to the Parrot AR.Drone 2.0 from Delphi XE5**. 2014. Disponível em: <http://delphi.org/2014/02/connecting-to-the-parrot-ar-drone-2-0-from-delphi-xe5/>. Acesso em: 12 abr. 2015.

MOI, Alberto; MOREIRA, Rodrigo Couto; GEREMIA, Marina. Metodos matemáticos para definição de posicionamento. **Ágora: Revista eletrônica,** Cerro Grande, n. 19, p.69-79, dez. 2014. Mensal. Disponível em: <http://agora.ceedo.com.br/ojs/index.php/AGORA_Revista_Eletronica/article/viewFile/167/161>. Acesso em: 26 maio 2016.

O'KANE, M. Jason. **A Gentle Introduction to ROS.** Carolina do Sul, Estados Unidos da America: 315 Main Street: Createspace Independent Publishing Platform, 2013. 166 p. Disponiível em http://www.cse.sc.edu/~jokane/agitr/.

PARROT. AR.Drone 2.0 Parrot. [Paris], 2014. Disponível em: <http://ardrone2.parrot.com/>. Acesso em 12 abr 2015.

PFEIFER, Erick. **Projeto e Controle de um UAV Quadrirotor**. 2013. 132 p. Dissertação de mestrado (Mestre em Engenharia) – Escola Politécnica da Universidade de São Paulo, São Paulo, Brasil.

PORTAL, V. João. **A Java Autopilot for Parrot A.R. Drone Designed with DiaSpec**. 2011. 46 p. Trabalho de Graduação (Engenharia da Computação) – Instituto de informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brasil.

RAMOS, C. Daniel. **Introdução ao ROS – Robot Operation System.** 2014. 14 f. Monografia (Especialização) - Curso de Engenharia de Automação e Sistemas, Centro Tecnológico, Universidade Federal de Santa Catarina, Florianópolis, 2014. Disponível em: <https://daniel8484.files.wordpress.com/2014/10/ros_intro_v1.pdf>. Acesso em: 20 fev. 2016.

ROS.org, **ROS.** 2007 Disponível em: < http://www.ros.org/about-ros/>. Acesso em: 28 mar 2015.

SÁ, C. Rejane. **Construção, modelagem dinâmica e controle PID para estabilidade de um veículo aéreo não tripulado do tipo quadrirotor**. 2012. 67 p. Dissertação de Mestrado (Engenharia de Teleinformática) – Centro de Tecnologia, Universidade Federal do Ceará, Ceará, Brasil.

SANTOS, Jorge Miguel Gonçalves Ferreira dos. **Avaliação da Qualidade do Posicionamento da Rede GNSS SERVIR - CIGeoE.** 2015. 140 f. Dissertação (Mestrado) - Curso de Engenharia Geográfica, Departamento de Engenharia GeogrÁfica, GeofÍsica e Energia, Universidade de Lisboa, Lisboa, 2015.

SANTOS, Landerson Gomes dos. **Aplicação Delphi utilizando Google Maps - Google Maps Library.** 2009 Disponível em: <http://landerson6.blogspot.com.br/2013/03/aplicacao-delphi-utilizando-google-maps.html>. Acesso em: 18 mar. 2016.

SOUZA, Erica Fernanda de. **Estudo para rastreamento de irradiadores de gamagrafia industrial por meio do uso do sistema de posicionamento global (GPS).** 2015. 42 f. Dissertação (Mestrado) - Curso de Energia Nuclear, Instituto Alberto Luiz Coimbra de Pós-graduação e Pesquisa de Engenharia, Rio de Janeiro, 2015.

VARELLA, João, **Os drones invadem os negócios.** 2014 Disponível em: < http://www.istoedinheiro.com.br/noticias/mercado-digital/20140124/drones-invadem-negocios/146050.shtml>. Acesso em: 13 abr 2015.

APÊNDICE A – Estruturas mapeadas para Delphi.

Nos Quadro 21 a 49 são apresentas todas as estruturas mapeadas utilizadas pelo ArDrone para envio de seus dados.

Quadro 21 – Estruturas mapeadas

```
1   ARDRONE_NAVDATA = record
2       // 3x3 matrix
3       matrix33_t : record
4         m11 : Single;
5         m12 : Single;
6         m13 : Single;
7         m21 : Single;
8         m22 : Single;
9         m23 : Single;
10        m31 : Single;
11        m32 : Single;
12        m33 : Single;
13      end;
14      // 3x1 vector
15      vector31_t : record
16          v : array [1..3] of Single;
17          pos : record
18              x : Single;
19              y : Single;
20              z : Single;
21          end;
22      end;
23  end;
```

Quadro 22 – Matrizes

```
1   // 3x3 matrix
2   matrix33_t = record
3     m11 : Single;
4     m12 : Single;
5     m13 : Single;
6     m21 : Single;
7     m22 : Single;
8     m23 : Single;
9     m31 : Single;
10    m32 : Single;
11    m33 : Single;
12  end;
13  // 3x1 vector
14  vector31_t = record
15      v : array [1..3] of Single;
16      pos : record
17          x : Single;
18          y : Single;
19          z : Single;
20      end;
21  end;
22  // 3x1 vector
23  vector21_t = record
24      v : array [1..2] of Single;
25      pos : record
26          x : Single;
27          y : Single;
28      end;
29  end;
```

Quadro 23 – Velocidades

```
1   // Velocities
2   velocities_t = record
3         x : Single;
4         y : Single;
5         z : Single;
6   end;
7
```

Quadro 24 – Navdata_demo

```
NAVDATA_DEMO = record
    tag : Word;
    size : Word;
    ctrl_state: cardinal;
    vbat_flying_percentage: cardinal;
    theta: Single;
    phi: Single;
    psi: Single;
    altitude: Integer;
    vx: Integer;
    vy: Integer;
    vz: Integer;
    num_frames: cardinal;                 // Don't use
    detection_camera_rot: vector31_t;       // Don't use
    detection_camera_trans: vector31_t;     // Don't use
    detection_tag_index: cardinal;        // Don't use
    detection_camera_type: cardinal;      // Don't use
    drone_camera_rot: vector31_t;            // Don't use
    drone_camera_trans: vector31_t;         // Don't use
end;
```

Quadro 25 – Navdat_time

```
NAVDATA_TIME = record
        tag: Word;
        size: Word;
        time: cardinal;
  end;// time;
```

Quadro 26 – Navdat_Raw_Measures

```
  // Raw measurements
NAVDATA_RAW_MEASURES = record
      tag: Word;
      size: Word;

      raw_accs: array [0..2] of Word ; // filtered accelerometers
      raw_gyros: array [0..2] of smallint; // filtered gyrometers
      aw_gyros_110: array [0..1] of smallint; // gyrometers  x/y 110 deg/s

      vbat_raw: cardinal;             // battery voltage raw (mV)
      us_debut_echo: Word;
      us_fin_echo: Word;
      us_association_echo: Word;
      us_distance_echo: Word;
      us_courbe_temps: Word;
      us_courbe_valeur: Word;
      us_courbe_ref: Word;
      flag_echo_ini: Word;
      //Word frame_number;
      nb_echo: Word;
      sum_echo: cardinal;
      alt_temp_raw: integer;
      gradient: smallint;
 end;// raw_measures;
```

Quadro 27 – Navdat_Phys_Measures

```
1     // Physical measurements
2    NAVDATA_PHYS_MEASURES = record
3         tag: Word;
4         size: Word;
5         accs_temp: single;
6         gyro_temp: Word;
7         phys_accs: array [0..2] of single;
8         phys_gyros: array [0..2] of single;
9         alim3V3: cardinal;          // 3.3 volt alim       [LSB]
10        vrefEpson: cardinal;        // ref volt Epson gyro [LSB]
11        vrefIDG: cardinal;          // ref volt IDG gyro   [LSB]
12     end;// phys_measures;
13
```

Quadro 28 – Navdat_Gyros_OffSets

```
1     // Gyros offsets
2     NAVDATA_GYROS_OFFSETS = record
3         tag: Word;
4         size: Word;
5         offset_g: array [0..2] of single;
6     end;// gyros_offsets;
7
```

Quadro 29 – Navdat_Eules_Angles

```
1     // Euler angles
2     NAVDATA_EULER_ANGLES = record
3         tag: Word;
4         size: Word;
5         theta_a: single;
6         phi_a: single;
7     end;// euler_angles;
```

Quadro 30 – Navdat_References

```
1     // References
2     NAVDATA_REFERENCES = record
3         tag: Word;
4         size: Word;
5        ref_theta: integer;
6        ref_phi: integer;
7        ref_theta_I: integer;
8        ref_phi_I: integer;
9        ref_pitch: integer;
10       ref_roll: integer;
11       ref_yaw: integer;
12       ref_psi: integer;
13       vx_ref: single;
14        vy_ref: single;
15        theta_mod: single;
16        phi_mod: single;
17        k_v_x: single;
18        k_v_y: single;
19        k_mode: cardinal;
20        ui_time: single;
21        ui_theta: single;
22        ui_phi: single;
23        ui_psi: single;
24        ui_psi_accuracy: single;
25       ui_seq: integer;
26    end;//references;
27
```

Quadro 31 – Navdat_Trims

```
1     // Trims
2     NAVDATA_TRIMS = record
3         tag: Word;
4         size: Word;
5         angular_rates_trim_r: single;
6         euler_angles_trim_theta: single;
7         euler_angles_trim_phi: single;
8     end;// trims;
```

Quadro 32 – Navdat_RC_References

```
1     // RC references
2     NAVDATA_RC_REFERENCES = record
3         tag: Word;
4         size: Word;
5        rc_ref_pitch: integer;
6        rc_ref_roll: integer;
7        rc_ref_yaw: integer;
8        rc_ref_gaz: integer;
9        rc_ref_ag: integer;
10    end;// rc_references;
```

Quadro 33 – Navdat_PWM

```
1     // PWM
2     NAVDATA_PWM = record
3         tag: Word;
4         size: Word;
5         motor1: byte;
6         motor2: byte;
7         motor3: byte;
8         motor4: byte;
9         sat_motor1: byte;
10        sat_motor2: byte;
11        sat_motor3: byte;
12        sat_motor4: byte;
13       gaz_feed_forward: single;
14       gaz_altitude: single;
15       altitude_integral: single;
16       vz_ref: single;
17       u_pitch: integer;
18       u_roll: integer;
19       u_yaw: integer;
20       yaw_u_I: single;
21       u_pitch_planif: integer;
22       u_roll_planif: integer;
23       u_yaw_planif: integer;
24        u_gaz_planif: single;
25        current_motor1: Word;
26        current_motor2: Word;
27        current_motor3: Word;
28        current_motor4: Word;
29        altitude_prop: single;
30        altitude_der: single;
31    end;// pwm;
```

Quadro 34 – Navdat_Altitude

```
1     // Altitude
2     NAVDATA_ALTITUDE = record
3         tag: Word;
4         size: Word;
5        altitude_vision: integer;
6        altitude_vz: single;
7        altitude_ref: integer;
8        altitude_raw: integer;
9        obs_accZ: single;
10       obs_alt: single;
11       obs_x: vector31_t;
12       obs_state: cardinal;
13        est_vb: vector21_t;
14        est_state: cardinal;
15    end;// altitude;
16
```

Quadro 35 – Navdat_Vision_Raw

```
1       // Vision (raw)
2       NAVDATA_VISION_RAW = record
3           tag: Word;
4           size: Word;
5           vision_tx_raw: single;
6           vision_ty_raw: single;
7           vision_tz_raw: single;
8       end;// vision_raw;
9
```

Quadro 36 – Navdat_Vision_OF

```
1       // Vision (offset?)
2       NAVDATA_VISION_OF = record
3           tag: Word;
4           size: Word;
5           of_dx: array [0..4] of single;
6           of_dy: array [0..4] of single;
7       end;//vision_of;
8
```

Quadro 37 – Navdat_Vision

```
1     // Vision
2     NAVDATA_VISION = record
3         tag: Word;
4         size: Word;
5         vision_state: cardinal;
6        vision_misc: integer;
7        vision_phi_trim: single;
8        vision_phi_ref_prop: single;
9        vision_theta_trim: single;
10       vision_theta_ref_prop: single;
11       new_raw_picture: integer;
12       theta_capture: single;
13       phi_capture: single;
14       psi_capture: single;
15       altitude_capture: integer;
16       time_capture: cardinal;    // time in TSECDEC format (see config.h)
17       body_v: velocities_t;
18       delta_phi: single;
19       delta_theta: single;
20       delta_psi: single;
21       gold_defined: cardinal;
22       gold_reset: cardinal;
23       gold_x: single;
24       gold_y: single;
25    end;// vision;
```

Quadro 38 – Vision_Perf

```
1     // Vision performances
2     NAVDATA_VISION_PERF = record
3         tag: Word;
4         size: Word;
5         time_szo: single;
6         time_corners: single;
7         time_compute: single;
8         time_tracking: single;
9         time_trans: single;
10        time_update: single;
11        time_custom: array [0..19] of single;
12    end;// vision_perf;
```

Quadro 39 – Screen_Point_t

```
1    // Screen point
2    screen_point_t = record
3        x: integer;
4        y: integer;
5    end;
```

Quadro 40 – Navdat_Trackers_Send

```
1    // Trackers
2    NAVDATA_TRACKERS_SEND = record
3        tag: Word;
4        size: Word;
5       locked: array [0..29] of integer;
6       point: array [0..29] of screen_point_t;
7    end;// trackers_send;
8
```

Quadro 41 – Navdat_Vision_Detect

```
1    // Vision detection
2    NAVDATA_VISION_DETECT = record
3        tag: Word;
4        size: Word;
5        nb_detected: cardinal;
6        type_ :array[1..3] of cardinal;
7        xc: array[1..3] of cardinal;
8        yc: array[1..3] of cardinal;
9        width: array[1..3] of cardinal;
10       height: array[1..3] of cardinal;
11       dist: array[1..3] of cardinal;
12       orientation_angle: array[1..3] of single;
13       rotation: array[1..3] of matrix33_t;
14       translation: array[1..3] of vector31_t;
15       camera_source: array[1..3] of cardinal;
16   end;//vision_detect;
```

Quadro 42 – Navdat_Watchdog

```
1    // Watchdog
2    NAVDATA_WATCHDOG = record
3        tag: Word;
4        size: Word;
5       watchdog: integer;
6    end;// watchdog;
7
```

Quadro 43 – ADC_Data_Frame

```
1    // ADC data
2    NAVDATA_ADC_DATA_FRAME = record
3        tag: Word;
4        size: Word;
5        version: cardinal;
6        data_frame: array [0..31] of byte;
7    end;//adc_data_frame;
8
```

Quadro 44 – Navdat_Video_Stream

```
    // Video stream
    NAVDATA_VIDEO_STREAM = record
        tag: Word;
        size: Word;
        quant: byte; // quantizer reference used to encode frame [1:31]
        frame_size: cardinal;          // frame size (bytes)
        frame_number: cardinal;        // frame index
        atcmd_ref_seq: cardinal;       // atmcd ref sequence number
        atcmd_mean_ref_gap: cardinal; // mean time between two consecutive
atcmd_ref (ms)
        atcmd_var_ref_gap: single;
        atcmd_ref_quality: cardinal;   // estimator of atcmd link quality

        // drone2
        out_bitrate: cardinal; // measured out throughput from the video tcp
socket
        desired_bitrate: cardinal;| // last frame size generated by the video
encoder
       data1: integer;
       data2: integer;
       data3: integer;
       data4: integer;
       data5: integer;
       tcp_queue_level: cardinal;
       fifo_queue_level: cardinal;
    end;//video_stream;
```

Quadro 45 – Navdat_Games

```
    // Games
    NAVDATA_GAMES = record
        tag: Word;
        size: Word;
        double_tap_counter: cardinal;
        finish_line_counter: cardinal;
    end;// games;

```

Quadro 46 – Navdat_Pressure_Raw

```
    // Preassure (raw)
    NAVDATA_PRESSURE_RAW = record
        tag: Word;
        size: Word;
        up: cardinal;
        ut: Word;
        temperature_meas: cardinal;
        pression_meas: cardinal;
    end;//} pressure_raw;

```

Quadro 47 – Navdat_Magneto

```
1     // Magneto
2     NAVDATA_MAGNETO = record //coordenadas polares
3         tag: Word;
4         size: Word;
5         mx: smallint;
6         my: smallint;
7         mz: smallint;
8         magneto_raw: vector31_t;// magneto in the body frame, in mG
9         magneto_rectified: vector31_t;
10        magneto_offset: vector31_t;
11        heading_unwrapped: single;
12        heading_gyro_unwrapped: single;
13        heading_fusion_unwrapped: single;
14        magneto_calibration_ok: byte;
15        magneto_state: cardinal;
16        magneto_radius: single;
17        error_mean: single;
18        error_var: single;
19        tmp1: single;
20        tmp2: single;
21    end;// magneto;
```

Quadro 48 – TChannels

```
1   TChannels = record
2     sat: byte;
3     cn0: byte;
4   end;
```

Quadro 49 – TNavData

```
1   TNavData = record
2     header : Integer;
3     ardrone_state: Integer;
4     sequence: Integer;
5     vision_defined: Integer;
6   end;
7
```

Quadro 50 – Navdat_GPS

```
 NAVDATA_GPS = record
        tag: Word;
        size: Word;
        lat: double;                    //*!< Latitude */
        lon: double;                    //*!< Longitude */
        double: double;               //*!< Elevation */
        hdop: double;                   //*!< hdop */
        data_available : cardinal;       //*!< When there is data
available */
        unk_0: array [0..7] of byte;
        lat0:double;                   //*!< Latitude ??? */
        lon0:double;                   //*!< Longitude ??? */
        lat_fuse:double;               //*!< Latitude fused */
        lon_fuse:double;               //*!< Longitude fused */
        gps_state: cardinal;//*!< State of the GPS, still need to
figure out */
        unk_1:  array [0..39] of byte;
        vdop:double;                   //*!< vdop */
        pdop:double;                   //*!< pdop */
        speed: Single;                  //*!< speed */
        last_frame_timestamp: cardinal; //*!< Timestamp from the
last frame */
        degree: Single;                 //*!< Degree */
        degree_mag: Single;             //*!< Degree of the magnetic
*/
        unk_2: array [0..15] of byte;
        channels: array [0..11] of TChannels;
        gps_plugged: integer;           //*!< When the gps is plugged
*/
        unk_3: array [0..107] of byte;
        gps_time: double;              //*!< The gps time of week */
        week  : Word;                  //*!< The gps week */
        gps_fix: byte;               //*!< The gps fix */
        num_sattelites: byte;       //*!< Number of sattelites */
        unk_4:  array [0..23] of byte;
        ned_vel_c0:double;            //*!< NED velocity */
        ned_vel_c1:double;            //*!< NED velocity */
        ned_vel_c2:double;            //*!< NED velocity */
        pos_accur_c0:double;          //*!< Position accuracy */
        pos_accur_c1:double;          //*!< Position accuracy */
        pos_accur_c2:double;          //*!< Position accuracy */
        speed_acur: Single;            //*!< Speed accuracy */
        time_acur: Single;             //*!< Time accuracy */
        unk_5: array [0..71] of byte;
        temprature: Single;
        pressure: Single;
  end;
```