Universidade Lusófona de Humanidades e Tecnologias

Licenciatura em Engenharia Informática

Relatório do Trabalho Final de Curso

Vigilância Activa em Robótica Móvel

Autores	Bruno David Francisco Miguel nº20064160
	Manuel João dos Santos Gomes nº20063290
Orientador	Prof. Sérgio Guerreiro

Lisboa, Dezembro de 2008

Índice

Abstract

This paper describes an Autonomous Mobile Robot project, developed by two students from the Computer Engineering course. A mobile robot as to had the ability to interact with the environment and has the function of surveillance, of a specific area known or an area that the robot had to learn while is interacting with it.

This paper describes the robot solution adopted by this team and the respective improvements that could turn this solution most appropriate to this matter.

1 Introdução

Em robótica móvel existem três questões chaves:

- Aonde estou?
- Aonde vou?
- Como chegar lá?

Para responder a estas questões o robot tem que, ter o modelo do ambiente (dado ou construído autonomamente), observar e analisar o ambiente, encontrar a sua posição no ambiente e planear e executar o movimento a efectuar.

Um robot móvel tem a capacidade para se mover no mundo real e ser completamente autónomo. Podendo também ser usado para fazer tarefas específicas, como policiamento, trabalhos em armazéns, hospitais, etc.

1.1 Objectivo

De um modo geral este projecto tem por objectivo simular num ambiente real, um robot móvel que tem como tarefa principal visitar três cones existentes nesse mesmo ambiente. Este projecto contém 2 módulos caracterizados por Árbitro e Simulação. O módulo Árbitro lida com as questões da definição do ambiente em si, definindo os obstáculos e suas posições, a cor do chão, a luz no ambiente simulando o sol e regula as movimentações do robot, fazendo com que o robot só acabe a sua tarefa depois de ter visitado os cones na sua totalidade, independentemente da ordem. Este módulo indica também ao robot quais as posições onde os cones se encontram no ambiente, podendo estas ser posições dadas por um GPS. O módulo Simulação define as movimentações do robot no ambiente e a forma como identificar os cones a visitar. As movimentações neste módulo são reguladas através de uma quantificação de contentamento, que traduz o caminho correcto no qual o robot deve seguir para visitar um dos cones, ou seja,

quanto maior for o grau de contentamento, mais esse caminho se encontra perto de um dos cones. Este grau é medido em todos os movimentos efectuados pelo robot.

Este módulo regula também a forma como o robot através da sua câmara, encontra os cones no ambiente simulado. Isto é feito pela cor dos cones, ou seja, em cada frame captada pela câmara são analisados os pixéis, quando são encontradas pixéis da mesma cor que correspondem aos cones, o robot segue o caminho até ao cone analisando frame a frame, calculando a distância a que se encontra e por onde deve seguir até o visitar.

O objectivo deste documento é descrever o trabalho realizado no desenvolvimento deste projecto, identificando possíveis falhas dando a conhecer possíveis soluções para tornar esta solução mais perfeita.

2 Descrição do Projecto

O projecto passou por várias fases que em conjunto permitiram chegar ao produto final. Essas fases passaram:

- Pelo estudo das tecnologias envolvidas para a realização do projecto;
- Pela análise das funções e requisitos que o projecto teria que ter;
- Pelo desenvolvimento do projecto em si;
- Pela realização do relatório do mesmo.

Para o desenvolvimento deste projecto foi utilizado como plataforma uma nova tecnologia da Microsoft, o Microsoft Robotics Studio 1.5.

O Microsoft Robotics Studio é um novo ambiente Windows para programadores amadores, académicos e comerciais, que simplifica o desenvolvimento de aplicações, nas quais podem ser usadas numa variedade de plataformas robóticas, desde os equipamentos reais aos simulados. Este ambiente apresenta vários benefícios: plataforma de desenvolvimento end-to-end que pode ser expandida facilmente, runtime orientado a serviços, etc.

A linguagem de programação escolhida para o desenvolvimento do mesmo foi C# (C-Sharp).

Esta linguagem é uma linguagem de programação orientada a objectos, desenvolvida pela Microsoft que faz parte da plataforma .Net. A sintaxe da mesma foi baseada no C++, mas inclui várias influências de outras linguagens como Java e Delphi.

2.1 Cronograma

A tabela que se segue mostra o período de tempo que cada fase requereu no tempo total para o desenvolvimento deste projecto.

Fases	Setembro	Outubro	Novembro
Estudo das Tecnologias			
Análise			
Desenvolvimento			
Relatório			

3 Análise

Em uma análise geral ao projecto, podemos identificar 8 estados a que o robot é submetido na realização da sua tarefa:

- **Reset;**
- **Pronto;**
- **Procura;**
- **Aproximacao;**
- **AproximacaoFinal;**
- **Afasta;**
- **EvitaColisao;**
- **Concluido.**

Por conseguinte, cada estado tem como objectivo realizar a seguinte função:

3.1 Reset

Sempre que se inicia a aplicação para que o robot fique pronto a fazer a sua tarefa é necessário pressionar o botão reset que se encontra na consola.

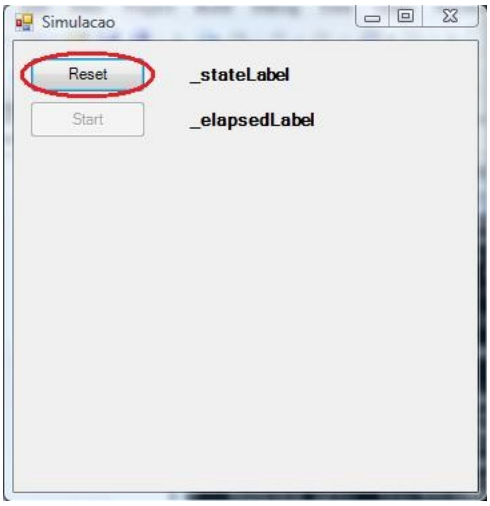


Fig.1 – Consola

Quando se prime o botão reset pela primeira vez começa-se a processar as frames da câmara do robot, mostra o tempo decorrido, acede ao serviço que manipula o sensor IR (infra-vermelhos) e acede ao serviço que manipula o acesso às operações principais de manuseamento de mensagens, através do serviço *Microsoft.Dss.ServiceModel.DsspServiceBase*.

```
if (!_iteratorsStarted)
{
    // começa a processar as frames da câmara
    SpawnIterator<DateTime>(DateTime.Now, ProcessFrame);

    // começa a mostrar o tempo decorrido
    SpawnIterator<DateTime>(DateTime.Now, UpdateElapsedTime);

    // acede ao serviço sensor IR
    _irSensorPort.Subscribe(_irNotifyPort);

    // acede ao serviço que manipula as mensagens
    MainPortInterleave.CombineWith(new Interleave(
        new TeardownReceiverGroup(),
        new ExclusiveReceiverGroup(),
        new ConcurrentReceiverGroup
        (
            Arbiter.Receive<irsensor.Replace>(true, _irNotifyPort, IRNotifyReplaceHandler)
        )
    ));

    // evita a começo de iterações adicionais quando se reinicia o cenário

    _iteratorsStarted = true;
}
```

Quando este botão é premido durante a simulação, toda a simulação é reiniciada através da instância *Arbiter* onde ajusta os portos para a comunicação com o serviço árbitro, inicia o serviço *Drive*, ajusta a velocidade do robot para 0, inicia o serviço *QuadDrive* existente neste tipo de robot (*Corobot*) e faz com que o robot passe ao estado Pronto.

```
yield return Arbiter.Choice(_refereePort.Get(),
    delegate(referee.MagellanRefereeState state) { SetWaypoints(state); },
    delegate(Fault f) { }
    );

_drivePort.EnableDrive(true);
_drivePort.SetDriveSpeed(0, 0);
_quadDrivePort.SetPose(new quadDrive.SetPoseRequestType(new Microsoft.Robotics.PhysicalModel.Proxy.Pose()));
_state.CurrentMode = ModeType.Pronto;
```

3.2 Pronto

Quando se pressiona o botão reset pela primeira vez, o robot fica pronto para efectuar a sua tarefa. Isto acontece pois acede-se à instância *Arbiter* e ao seu método *Receive*, para que a aplicação fique pronta para a manipulação de mensagens.

```
case ModeType.Pronto:
    yield return Arbiter.Receive(false, TimeoutPort(100), delegate(DateTime timeout) { });
    break;
```

Quando nos encontramos neste estado a visão que temos da consola é a seguinte:

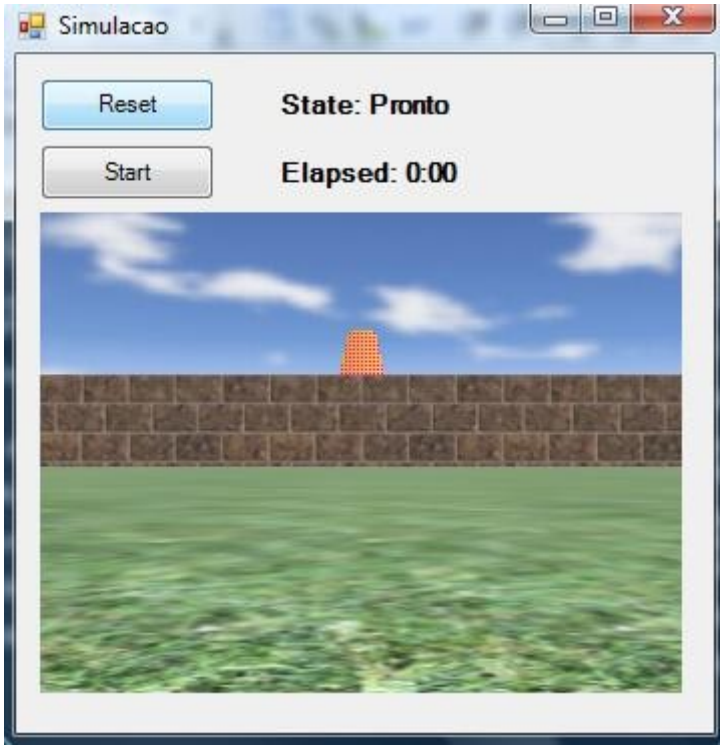


Fig.2 – Consola no estado Pronto

3.3 Procura

Neste estado a primeira coisa que é verificada é se o robot identificou algum dos cones a visitar através da sua câmara, se verificar que um cone é identificado então o robot passa ao estado Aproximacao.

```
case ModeType.Procura:
    {
        // verifica se foi identificado um cone
        if (_imageProcessResult != null)
        {
            if (_imageProcessResult.Area > _imageProcessResult.AreaThreshold)
            {
                // quando existe um cone na mira
                _state.CurrentMode = ModeType.Aproximacao;
                continue;
            }
        }
```

Se não for identificado nenhum cone então o robot vai começar a navegar no ambiente definido para as posições dos cones. As posições dos cones são dadas pelo módulo Árbitro. O robot para se movimentar para estas posições, desviando-se dos obstáculos, é necessário aceder à instância *Arbiter* onde por sua vez ela controla o serviço *QuadDrive* que vai fazer com que ele se movimente.

```
quadDrive.DriveDifferentialFourWheelState quadDriveState = null;
yield return Arbiter.Choice(_quadDrivePort.Get(),
    delegate(quadDrive.DriveDifferentialFourWheelState state)
    {
        quadDriveState = state;
    },
    delegate(Fault f) { }
);
```

A movimentação do robot é feita através da comparação da sua posição com as posições dos cones, ou seja, na movimentação do robot existe um estado de contentamento do mesmo que corresponde à sua posição. Quanto mais o robot estiver perto de um dado cone, maior é o seu grau de contentamento.

Sempre que o robot arranca na sua tarefa o seu grau de contentamento actual é o da sua posição inicial.

```
if (quadDriveState == null)
    continue;

double distance;
double degrees;
float currentHappiness = GetHappiness(quadDriveState.Position);
```

Sempre que o nível de contentamento actual for maior ao nível de contentamento anterior, o robot segue na mesma direcção dada uma distância e grau de direcção.

```
if (currentHappiness > _previousHappiness)
{
    // continua na mesma direcção
    distance = 0.25;
    degrees = 0;
}
```

Sempre que isso não acontecer o robot vai voltar atrás e vai tentar de novo, isto é feito mais uma vez acedendo à instância *Arbiter* que através do seu método *Choice* que por sua vez acede ao serviço *Drive* que faz com que o robot recue a distância que percorreu anteriormente. Em seguida é dado a distância a percorrer e é escolhido um ângulo entre -45 e -90 graus e reinicia o nível de contentamento para a posição anterior.

```
    // volta atrás e tenta de novo
    if(!driveCommandFailed)
        yield return Arbiter.Choice(
            _drivePort.DriveDistance(-0.25, _driveSpeed),
            delegate(DefaultUpdateResponseType response) { },
            delegate(Fault f) { driveCommandFailed = true; }
        );

    distance = 0.25;
    // escolhe um anglo entre -45 e -90
    degrees = (float)_random.NextDouble() * -45f - 45f;
    // reinicia o nivel de contentamento para a posição anterior
    _previousHappiness = _previousPreviousHappiness;
}
```

Sempre que o ângulo escolhido é diferente de 0, isto significa que o robot terá que fazer a rotação para o ângulo escolhido, isto é conseguido através do acesso à instância *Arbiter* e ao método *Choice* que acede ao serviço *Drive* e é-lhe ordenado que faça a rotação dos graus dados e que se movimente a distância dada.

```
if (degrees != 0)
{
    if (!driveCommandFailed)
        yield return Arbiter.Choice(
            _drivePort.RotateDegrees(degrees, _rotateSpeed),
            delegate(DefaultUpdateResponseType response) { },
            delegate(Fault f) { driveCommandFailed = true; }
        );
}

if (!driveCommandFailed)
    yield return Arbiter.Choice(
        _drivePort.DriveDistance(distance, _driveSpeed),
        delegate(DefaultUpdateResponseType response) { },
        delegate(Fault f) { driveCommandFailed = true; }
    );
```

Em seguida é atribuído ao nível anterior de contentamento a posição a que o robot se encontrava antes de se movimentar e voltar atrás.

```
    _previousPreviousHappiness = _previousHappiness;
    _previousHappiness = currentHappiness;

    break;
}
```

3.4 Aproximacao

O robot sempre que encontra um cone e fixa-o através da sua câmara e entra no estado Aproximacao.

Este estado verifica a leitura do sensor IR (infra-vermelhos) e através da instância *Arbiter* e o seu método *Choice* acede ao serviço do sensor IR e mede a distância da leitura do sensor IR.

```
case ModeType.Aproximacao:
    {
        float IRDistance = -1f;

        // verifica a leitura do sensor IR
        irsensor.Get tmp = new irsensor.Get();
        _irSensorPort.Post(tmp);
        yield return Arbiter.Choice(tmp.ResponsePort,
            delegate(irsensor.AnalogSensorState state)
            {
                if (state.NormalizedMeasurement < 1f)
                    IRDistance = (float)state.RawMeasurement;
            },
            delegate(Fault f) { }
            );
```

Sempre que a distância da leitura do sensor IR for maior ou igual a 0, então o robot passa ao estado AproximacaoFinal.

```
        if (IRDistance >= 0)
        {
            // confia no sensor IR para a aproximação final
            _state.CurrentMode = ModeType.AproximacaoFinal;
            break;
        }
```

Sempre que o processamento da imagem por parte da câmara seja nula, ou seja, não tenha identificado nenhum cone ou a área da imagem resultante seja menor que a área limiar da imagem resultante, então o robot volta ao estado Procura.

```
if ((_imageProcessResult == null) ||
    (_imageProcessResult.Area < _imageProcessResult.AreaThreshold))
{
    // se perde cone, volta ao modo procura
    _state.CurrentMode = ModeType.Procura;
    continue;
```

Caso contrário se o robot tiver o cone fixado na imagem resultante pela sua câmara, o robot vai fazer a aproximação ao cone através de um anglo que é calculado através do que é necessário rodar para que a imagem do cone fique no centro da mira e a distância a percorrer dada. Com os valores do ângulo e da distância é necessário através da instância *Arbiter* e do seu método *Choice* aceder ao serviço *Drive* para que o robot faça a rotação pelo anglo calculado e percorra a distância dada.

```
    }
    float angle = _imageProcessResult.RightFromCenter * -2f * 5f;
    float distance = 0.25f;
    if (!driveCommandFailed)
        yield return Arbiter.Choice(
            _drivePort.RotateDegrees(angle, _rotateSpeed),
            delegate(DefaultUpdateResponseType response) { },
            delegate(Fault f) { driveCommandFailed = true; }
        );

    if (!driveCommandFailed)
        yield return Arbiter.Choice(
            _drivePort.DriveDistance(distance, _driveSpeed),
            delegate(DefaultUpdateResponseType response) { },
            delegate(Fault f) { driveCommandFailed = true; }
        );
    break;
}
```

3.5 AproximacaoFinal

Tal como no estado anterior é feita a leitura do sensor IR (infra-vermelhos) e mais uma vez através da instância *Arbiter* e do seu método *Choice* acede ao serviço do sensor IR e mede a distância até ao cone feita pelo sensor.

```
case ModeType.AproximacaoFinal:
    {
        // confia no sensor IR para a aproximação do cone
        float IRDistance = -1f;

        // leitura do sensor IR
        irsensor.Get tmp = new irsensor.Get();
        _irSensorPort.Post(tmp);
        yield return Arbiter.Choice(tmp.ResponsePort,
            delegate(irsensor.AnalogSensorState state)
            {
                if (state.NormalizedMeasurement < 1f)
                    IRDistance = (float)state.RawMeasurement;
            },
            delegate(Fault f) { }
            );
```

Sempre que a distância calculada for menor que 0, ou seja, que o robot tenha perdido o cone da sua mira, ele volta ao estado Procura.

```
        if (IRDistance < 0)
        {
            // volta para a procura visual
            _state.CurrentMode = ModeType.Procura;
            break;
        }
```

Caso contrário o serviço *Drive* que é acedido através da instância *Arbiter* e do seu método *Choice*, faz com que o robot se movimente, com a distância calculada anteriormente pelo sensor IR e à velocidade pretendida.

Em seguida o serviço *QuadDrive* é acedido mais uma vez através da instância *Arbiter* onde é verificado o seu estado e onde é possível através do mesmo saber a posição do robot no ambiente simulado, sempre que o estado do serviço *QuadDrive* é diferente de

null é calculada uma distância através da diferença das coordenadas da posição do robot e das coordenadas dos cones, se essa distância for menor que 1 então esse cone é marcado como visitado, caso contrário o robot passa ao estado Afasta.

```
        // marca o cone como visitado
        quadDrive.DriveDifferentialFourWheelState quadDriveState = null;
        yield return Arbiter.Choice(_quadDrivePort.Get(),
            delegate(quadDrive.DriveDifferentialFourWheelState state)
            {
                quadDriveState = state;
            },
            delegate(Fault f) { }
        );

        if (quadDriveState != null)
        {
            for (int i = 0; i < _waypoints.Count; i++)
            {
                float dx = (quadDriveState.Position.X - _waypoints[i].Location.X);
                float dz = (quadDriveState.Position.Z - _waypoints[i].Location.Z);
                float distance = (float)Math.Sqrt(dx * dx + dz * dz);
                if (distance < 1f)
                    _waypoints[i].Visited = true;
            }
        }

        _state.CurrentMode = ModeType.Afasta;

        break;
    }
```

3.6 Afasta

Neste estado o robot volta atrás através da instância *Arbiter* e do seu método *Choice* que acede ao serviço *Drive* em que o robot se movimenta à distância pretendida e faz uma rotação de 45 graus à esquerda.

```
case ModeType.Afasta:
    {
        // volta atrás e dá a volta
        if (!driveCommandFailed)
            yield return Arbiter.Choice(
                _drivePort.DriveDistance(-1, _driveSpeed),
                delegate(DefaultUpdateResponseType response) { },
                delegate(Fault f) { driveCommandFailed = true; }
            );

        if (!driveCommandFailed)
            yield return Arbiter.Choice(
                _drivePort.RotateDegrees(45, _rotateSpeed),
                delegate(DefaultUpdateResponseType response) { },
                delegate(Fault f) { driveCommandFailed = true; }
            );
```

O robot passa ao estado Concluido se todos os cones foram visitados, se isso não se verificar ele volta ao estado Procura para visitar os cones restantes, se realmente todos os cones foram visitados e o robot se encontra no estado Concluido então o tempo decorrido pára de contar e temos o tempo total que o robot demorou a executar a sua tarefa.

```
_state.CurrentMode = ModeType.Concluido;
for (int i = 0; i < _waypoints.Count; i++)
{
    if (!_waypoints[i].Visited)
    {
        _state.CurrentMode = ModeType.Procura;
        _previousHappiness = 0;
        _previousPreviousHappiness = 0;
        break;
    }
}

if (_state.CurrentMode == ModeType.Concluido)
{
    // odos os cones foram visitados
    _endTime = DateTime.Now;
}

break;
```

3.7 EvitaColisao

Este estado existe no caso de o robot necessitar de se desviar de obstáculos enquanto realiza a sua tarefa. Sempre que o robot entra neste estado ele anda para trás, vira à esquerda, entra novamente no estado Procura e movimenta-se para a frente.

Isto é conseguido através do acesso ao serviço *Drive* do robot, utilizando a instância *Arbiter* e o método *Choice*, no caso da movimentação atrás basta indicar valores negativos na distância a percorrer, em relação à movimentação à esquerda indica-se o ângulo de rotação pretendido e para se movimentar para a frente indica-se valores positivos da distância a percorrer.

```
case ModeType.EvitaColisao:
    {
        // volta para trás
        if (!driveCommandFailed)
            yield return Arbiter.Choice(
                _drivePort.DriveDistance(-0.25, _driveSpeed),
                delegate(DefaultUpdateResponseType response) { },
                delegate(Fault f) { driveCommandFailed = true; }
            );

        // vira à esquerda
        if (!driveCommandFailed)
            yield return Arbiter.Choice(
                _drivePort.RotateDegrees(90, _rotateSpeed),
                delegate(DefaultUpdateResponseType response) { },
                delegate(Fault f) { driveCommandFailed = true; }
            );

        _state.CurrentMode = ModeType.Procura;

        // movimenta-se para a frente
        if (!driveCommandFailed)
            yield return Arbiter.Choice(
                _drivePort.DriveDistance(1, _driveSpeed),
                delegate(DefaultUpdateResponseType response) { },
                delegate(Fault f) { driveCommandFailed = true; }
            );
        break;
    }
```

3.8 Concluido

O robot entra neste estado sempre que todos os cones foram visitados. O robot entra em rotação de 180 graus sobre si mesmo para indicar que completou a sua tarefa. Isto é conseguido acedendo ao serviço *Drive*, mais uma vez através da instância *Arbiter* e do seu método *Choice* dando valores à rotação pretendida.

```
case ModeType.Concluido:
    if (!driveCommandFailed)
        yield return Arbiter.Choice(
            _drivePort.RotateDegrees(180, _rotateSpeed),
            delegate(DefaultUpdateResponseType response) { },
            delegate(Fault f) { driveCommandFailed = true; }
        );
    break;
```

4 Implementação

Na implementação deste projecto foi identificado alguns melhoramentos que poderiam ser implementados de forma a garantir, uma melhor prestação dos movimentos efectuados pelo robot na visita dos cones existentes no ambiente simulado, na medida em que garantisse uma visita mais rápida dos mesmos. Também para aproximar esta simulação de várias realidades possíveis, poderia ser implementado também a possibilidade de termos múltiplos robots.

4.1 Movimentos

Á medida que vamos utilizando esta simulação, vamos poder verificar que muitas das vezes o robot faz movimentos feitos já anteriormente, que não garantem o sucesso de visitar um dos cones, querendo isto dizer que quando o robot efectua um caminho que não tem sucesso na resolução da sua tarefa, não devesse ser percorrido outra vez pelo robot.

O robot poderia reconhecer os caminhos por onde já passou sabendo se optando por esse caminho o levará ao sucesso da sua tarefa. Os caminhos poderiam ser memorizados pelo robot e guardados de maneira a garantir que numa próxima vez que efectua-se a sua tarefa o faria mais rápido e eficaz.

Uma outra forma de garantir os movimentos do robot mais eficazes, seria implementar uma solução em que o robot seguisse uma linha desenhada no chão, que lhe indicasse os caminhos a seguir para visitar os cones, nesta implementação não poderíamos demonstrar o robot a desviar-se de obstáculos, pois não faz sentido ter linhas a indicar os caminhos e colocar obstáculos, assim o robot quando encontra-se um, não saberia

como contorná-lo e seguir a linha existente para a realização da sua tarefa. Estas linhas seriam colocadas por caminhos onde não existissem obstáculos.

4.2 Múltiplos Robots

Para aproximar esta simulação de certas realidades, como por exemplo, ter vários robots a cumprir a mesma tarefa é possível, através da implementação de outro robot ou mais, bastando que em todos os estados demonstrados anteriormente fossem acedidos os serviços de todos os robots nos estados pretendidos.

Para criar vários robots podemos no módulo Árbitro que gere o ambiente simulado aceder à função *PopulateWorld()* e criar os robots pretendidos. Em seguida é mostrado como é criado o robot utilizado na nossa simulação.

```
// cria um Corobot
SimulationEngine.GlobalInstancePort.Insert(new corobot.CorobotEntity("Corobot", new Vector3(0, 0, 0)));
```

Conclusão

Com a realização deste trabalho pudemos adquirir novos conhecimentos a nível da programação C# como também utilizar novas tecnologias da Microsoft.

Com utilização das tecnologias Microsoft Robotics Studio 1.5 e Visual Studio C# 2005 pudemos experimentar variados ambientes virtuais nomeadamente robôs na realização de diversas tarefas através de serviços que vêm predefinidos no Microsoft Robotics Studio 1.5.

Este tipo de simulação visa demonstrar que o Microsoft Robotics Studio é compatível com as aplicações, serviços e robots de diversas empresas actuais (ex: LEGO, CoroWare, iRobo, etc).

O Microsoft Robotics Studio inclui uma ferramenta de programação visual, facilitando a criação e depuração de aplicativos robotacionais, permitindo desenvolver e gerar serviços modulares para hardware e software, permitindo que os utilizadores interajam com robôs através de interfaces com base na Web ou do Windows. Com este software é possível realizar inúmeros testes, sem custos, nem riscos de se manipular um robô real e até possibilita programar um dispositivo que ainda não existe.

Com a realização deste projecto pretendemos demonstrar uma das possíveis aplicações que podem ser criadas a nível da segurança e vigilância activa na robótica móvel.

Bibliografia

http://robotics.ethz.ch/books/autonomousmobilerobots/

http://repositorium.sdum.uminho.pt/handle/1822/3109

http://lrm.isr.ist.utl.pt/lrm/index.html

http://personal.stevens.edu/~yguo1/EE631/Introduction.pdf

http://www.soft-tech.com.au/MSRS/Intro/Intro.htm

http://msdn.microsoft.com/en-us/robotics/default.aspx

http://www.promrds.com/

http://guerrarj.spaces.live.com/blog/cns!1D04FD448E6D38B1!592.entry?wa=wsignin1.0

http://pesquompile.wikidot.com/microsoft-robotics-studio

Anexo 1 – Código Fonte

- ## Árbitro

```
using Microsoft.Ccr.Core;
using Microsoft.Dss.Core;
using Microsoft.Dss.Core.Attributes;
using Microsoft.Dss.ServiceModel.Dssp;
using Microsoft.Dss.ServiceModel.DsspServiceBase;

using System;
using System.Collections.Generic;
using W3C.Soap;

using Microsoft.Robotics.Simulation;
using Microsoft.Robotics.Simulation.Engine;
using engineproxy = Microsoft.Robotics.Simulation.Engine.Proxy;
using Microsoft.Robotics.Simulation.Physics;
using Microsoft.Robotics.PhysicalModel;
using System.ComponentModel;

using xna = Microsoft.Xna.Framework;
using xnagrfx = Microsoft.Xna.Framework.Graphics;

using corobot = ProMRDS.Simulation.Corobot;

namespace ProMRDS.Simulation.MagellanReferee
{
    [DisplayName("Arbitro da Simulacao")]
    [Description("O serviço do arbitro para a simulacao.")]
    [Contract(Contract.Identifier)]
```

```
public class MagellanReferee : DsspServiceBase
{
    public Barrier[] Barriers = new Barrier[]
    {
        new Barrier("Wall0", new Vector3(0, 0, -4), new Vector3(4f, 0.8f, 0.1f),
"BrickWall.dds", new Quaternion(0,0,0,1)),
        new Barrier("Wall1", new Vector3(-2, 0, -10), new Vector3(4f, 0.8f, 0.1f),
"BrickWall.dds", new Quaternion(0,0,0,2)),
        new Barrier("Wall2", new Vector3(-2.05f, 0, -3.05f), new Vector3(2f, 0.8f, 0.1f),
"BrickWall.dds", Quaternion.FromAxisAngle(0, 1, 0, (float)(Math.PI / 2))),
        new Barrier("Wall3", new Vector3(1.41f, 0, 2.46f), new Vector3(6f, 0.8f, 0.1f),
"BrickWall.dds", Quaternion.FromAxisAngle(0, 1, 0, (float)(Math.PI / 2))),
        new Barrier("Wall4", new Vector3(-4.41f, 0, 2.46f), new Vector3(6f, 0.8f, 0.1f),
"BrickWall.dds", Quaternion.FromAxisAngle(0, 1, 0, (float)(Math.PI / 2))),
        new Barrier("Tower0", new Vector3(5.58f, 2f, -0.59f), new Vector3(2f, 4f, 2f),
"MayangConcrP.dds", new Quaternion(0,0,0,1)),
        new Barrier("Tower1", new Vector3(8.58f, 2f, 9.59f), new Vector3(2f, 4f, 2f),
"MayangConcrP.dds", new Quaternion(0,0,0,1)),
    };

    [Partner("Engine",
        Contract = engineproxy.Contract.Identifier,
        CreationPolicy = PartnerCreationPolicy.UseExistingOrCreate)]
    private engineproxy.SimulationEnginePort _engineStub =
        new engineproxy.SimulationEnginePort();

    [InitialStatePartner(Optional = true, ServiceUri = "Magellan.Referee.config.xml")]
    private MagellanRefereeState _state = new MagellanRefereeState();

    // Porto do serviço proncipal
    [ServicePort("/MagellanReferee", AllowMultipleInstances=false)]
    private MagellanRefereeOperations _mainPort =
        new MagellanRefereeOperations();
```

```
public MagellanReferee(DsspServiceCreationPort creationPort) :
        base(creationPort)
{

}

protected override void Start()
{
    // garante que estado é valido
    ValidateState();

    base.Start();

    // Acrescenta objectos no mundo simulado
    PopulateWorld();
}

private void ValidateState()
{
    if (_state == null)
        _state = new MagellanRefereeState();

    if ((_state.Cones == null) || (_state.Cones.Length == 0))
    {
        // estado base
        _state.Cones = new Vector3[]
        {
            new Vector3(0, 0, -5),
            new Vector3(10, 0, 4),
            new Vector3(-3, 0, 2)
        };
        base.SaveState(_state);
```

```
    }
}

private void PopulateWorld()
{
    // inicia vista inicial
    CameraView view = new CameraView();
    view.EyePosition = new Vector3(-0.91f, 0.67f, -1f);
    view.LookAtPoint = new Vector3(1.02f, 0.09f, 0.19f);
    SimulationEngine.GlobalInstancePort.Update(view);

    // cria um munod simples e acrescenta o Corobot
    // acrescenta outra camera com vista da cena de cima
    CameraEntity fromAbove = new CameraEntity(640, 480);
    fromAbove.State.Name = "FromAbove";
    fromAbove.Location = new xna.Vector3(4.3f, 20.59f, 0.86f);
    fromAbove.LookAt = new xna.Vector3(4.29f, 18.26f, 0.68f);
    fromAbove.IsRealTimeCamera = false;
    SimulationEngine.GlobalInstancePort.Insert(fromAbove);


    SkyDomeEntity sky = new SkyDomeEntity("skydome.dds", "sky_diff.dds");
    SimulationEngine.GlobalInstancePort.Insert(sky);

    // Adiciona uma luz direcional para simular o Sol.
    LightSourceEntity sun = new LightSourceEntity();
    sun.State.Name = "Sun";
    sun.Type = LightSourceEntityType.Directional;
    sun.Color = new Vector4(1, 1, 1, 1);
    sun.Direction = new Vector3(-0.47f, -0.8f, -0.36f);
    SimulationEngine.GlobalInstancePort.Insert(sun);

    HeightFieldShapeProperties hf = new HeightFieldShapeProperties("height field",
```

```
            64, // numero de linhas
            10, // distancia em metros, entre linhas
            64, // numero de colunas
            10, // distancia em metrso, entre colunas
            1, // factor escala para varios valores de altura
            -1000);


        hf.HeightSamples = new HeightFieldSample[hf.RowCount * hf.ColumnCount];
        for (int i = 0; i < hf.RowCount * hf.ColumnCount; i++)
        {
            hf.HeightSamples[i] = new HeightFieldSample();
            hf.HeightSamples[i].Height = (short)(Math.Sin(i * 0.01));
        }

        // cria o material para o mundo
        hf.Material = new MaterialProperties("ground", 0.8f, 0.5f, 0.8f);

        // insere a entidade chão na simulação e especifica a textura
        SimulationEngine.GlobalInstancePort.Insert(new HeightFieldEntity(hf,
"FieldGrass.dds"));

        // cria as barreiras
        foreach (Barrier bar in Barriers)
        {
            SingleShapeEntity wall =
                new SingleShapeEntity(
                    new BoxShape(
                        new BoxShapeProperties(
                            0,
                            new Pose(),
                            bar.Dimensions)),
                        bar.Position);
```

```
            wall.State.Pose.Orientation = bar.Orientation;
            wall.State.Name = bar.Name;
            wall.State.Assets.DefaultTexture = bar.Texture;
            SimulationEngine.GlobalInstancePort.Insert(wall);
        }

        // cria cones
        for (int coneCount = 0; coneCount < _state.Cones.Length; coneCount++)
            SimulationEngine.GlobalInstancePort.Insert(
                CreateTriggerCone("Cone" + coneCount.ToString(), _state.Cones[coneCount]));

        // cria um Corobot
        SimulationEngine.GlobalInstancePort.Insert(new corobot.CorobotEntity("Corobot", new
Vector3(0, 0, 0)));
    }

    private VisualEntity CreateTriggerCone(
        string name,
        Vector3 position)
    {
        Vector3 dimensions = new Vector3(0.35f, 0.7f, 0.35f);
        position.Y += dimensions.Y / 2f;
        SingleShapeEntity triggerShape = new SingleShapeEntity(
            new BoxShape(
                new BoxShapeProperties(
                    0,
                    new Pose(),
                    dimensions)),
            position);

        // usado para receber uma notificação quando o cone e visitado
        Port<Shape> conePort = new Port<Shape>();
```

```
            triggerShape.State.Name = name;
            triggerShape.State.Assets.Mesh = "street_cone.obj";
            triggerShape.BoxShape.State.Name = name;
            triggerShape.BoxShape.State.Advanced = new ShapeAdvancedProperties();

            triggerShape.BoxShape.State.Advanced.IsTrigger = true;
            triggerShape.BoxShape.State.Advanced.TriggerNotificationPort = conePort;
            triggerShape.State.MassDensity.Mass = 0;
            triggerShape.State.MassDensity.Density = 0;

            Activate(Arbiter.Receive<Shape>(false, conePort,
               delegate(Shape sh)
               {
                  Console.WriteLine("O " + name + " foi visitado.");
               }
            ));

            return triggerShape;
         }


         [ServiceHandler(ServiceHandlerBehavior.Concurrent)]
         public virtual IEnumerator<ITask> GetHandler(Get get)
         {
            get.ResponsePort.Post(_state);
            yield break;
         }
      }

      public struct Barrier
      {
         public string Name;
```

```
        public Vector3 Position;
        public Vector3 Dimensions;
        public string Texture;
        public Quaternion Orientation;
        public Barrier(string name, Vector3 position, Vector3 dimensions, string texture,
Quaternion orientation)
        {
            Name = name;
            Position = position;
            Dimensions = dimensions;
            Texture = texture;
            Orientation = orientation;
        }
    }

    public static class Contract
    {
        public const string Identifier =
"http://www.promrds.com/contracts/2007/08/MagellanReferee.html";
    }

    #region State

    [DataContract()]
    public class MagellanRefereeState
    {
        [DataMember]
        public Vector3[] Cones;
    }
    #endregion

    [ServicePort]
```

```
    public class MagellanRefereeOperations : PortSet<DsspDefaultLookup, DsspDefaultDrop,
Get>
    {
    }

    public class Get : Get<GetRequestType, PortSet<MagellanRefereeState, Fault>>
    {
    }

}
```

- **Simulação**

```
using Microsoft.Ccr.Core;
using Microsoft.Ccr.Adapters.WinForms;
using Microsoft.Dss.Core;
using Microsoft.Dss.Core.Attributes;
using Microsoft.Dss.ServiceModel.Dssp;
using Microsoft.Dss.ServiceModel.DsspServiceBase;
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Xml;
using W3C.Soap;
using Microsoft.Robotics.PhysicalModel.Proxy;

using magellan = ProMRDS.Simulation.Magellan;
using drive = Microsoft.Robotics.Services.Drive.Proxy;
using quadDrive = ProMRDS.Simulation.QuadDifferentialDrive.Proxy;
using webcam = Microsoft.Robotics.Services.WebCam.Proxy;
using referee = ProMRDS.Simulation.MagellanReferee.Proxy;
using irsensor = Microsoft.Robotics.Services.AnalogSensor.Proxy;

namespace ProMRDS.Simulation.Magellan
{


    /// Implementação da classe Simulação

    [DisplayName("Simulacao")]
    [Description("Construção dos serviços para a simulação")]
    [Contract(Contract.Identifier)]
    public class SimMagellanService : DsspServiceBase
    {
```

```
        SimMagellanUI _magellanUI = null;

        // saber que cones foram visitados
        List<Waypoint> _waypoints = new List<Waypoint>();
        float _previousHappiness = 0;
        float _previousPreviousHappiness = 0;
        Random _random = new Random();

        // ultimo resultado do processamento de imagem
        ImageProcessResult _imageProcessResult = null;

        // comunicação com QuadDifferentialDrive
        [Partner(
            "QuadDrive",
            Contract = quadDrive.Contract.Identifier,
            CreationPolicy = PartnerCreationPolicy.UseExisting,
            Optional = false)]
        quadDrive.QuadDriveOperations _quadDrivePort = new
quadDrive.QuadDriveOperations();

        // comunicação com DifferentialDrive
        [Partner(
            "Drive",
            Contract = drive.Contract.Identifier,
            CreationPolicy = PartnerCreationPolicy.UseExisting,
            Optional = false)]
        drive.DriveOperations _drivePort = new drive.DriveOperations();

        // comunicação com a webcam do Corobot
        [Partner(
            "robotcam",
            Contract = webcam.Contract.Identifier,
            CreationPolicy = PartnerCreationPolicy.UseExisting,
```

```
            Optional = false)]
        webcam.WebCamOperations _cameraPort = new webcam.WebCamOperations();

        // comunicação com o serviço referee
        [Partner(
            "referee",
            Contract = referee.Contract.Identifier,
            CreationPolicy = PartnerCreationPolicy.UseExisting,
            Optional = false)]
        referee.MagellanRefereeOperations _refereePort = new
referee.MagellanRefereeOperations();

        // comunicação com o serviço IR sensor
        [Partner(
            "irsensor",
            Contract = irsensor.Contract.Identifier,
            CreationPolicy = PartnerCreationPolicy.UseExisting,
            Optional = false)]
        irsensor.AnalogSensorOperations _irSensorPort = new irsensor.AnalogSensorOperations();
        irsensor.AnalogSensorOperations _irNotifyPort = new irsensor.AnalogSensorOperations();

        // saber o tempo para o fim
        DateTime _startTime;
        DateTime _endTime;

        // Porto para receber enventos do interface de utilizador
        FromWinformEvents _fromWinformPort = new FromWinformEvents();


        // Estado
        private SimMagellanState _state = new SimMagellanState();

        // Porto principal
```

```
[ServicePort("/simmagellan", AllowMultipleInstances=false)]
private SimMagellanOperations _mainPort = new SimMagellanOperations();

// Construtor base de serviços

public SimMagellanService(DsspServiceCreationPort creationPort) :
        base(creationPort)
{
}

// Começo do serviço

protected override void Start()
{
   base.Start();

   // Inicia o manipulador de mensagens
   MainPortInterleave.CombineWith(new Interleave(
      new TeardownReceiverGroup(),
      new ExclusiveReceiverGroup
      (
         Arbiter.Receive<FromWinformMsg>(true, _fromWinformPort,
OnWinformMessageHandler)
      ),
      new ConcurrentReceiverGroup()
   ));

   // Criação do interface com o utilizador
   WinFormsServicePort.Post(new RunForm(CreateForm));

   _state.CurrentMode = ModeType.NotSpecified;
```

```
    SpawnIterator(BehaviorLoop);
}

// Cria o form UI
System.Windows.Forms.Form CreateForm()
{
    return new SimMagellanUI(_fromWinformPort);
}

// processamento de mensagens do UI
void OnWinformMessageHandler(FromWinformMsg msg)
{
    switch (msg.Command)
    {
        case FromWinformMsg.MsgEnum.Loaded:

            _magellanUI = (SimMagellanUI)msg.Object;
            break;

        case FromWinformMsg.MsgEnum.Reset:
            _state.CurrentMode = ModeType.Reset;
            break;

        case FromWinformMsg.MsgEnum.Start:
            if (_state.CurrentMode == ModeType.Pronto)
            {
                _startTime = DateTime.Now;
                _state.CurrentMode = ModeType.Procura;
            }
            break;
    }
}
```

```
// Inicia manipulador

[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public virtual IEnumerator<ITask> GetHandler(Get get)
{
   get.ResponsePort.Post(_state);
   yield break;
}

const float _rotateSpeed = 0.3f;
const float _driveSpeed = 0.5f;
bool _iteratorsStarted = false;

IEnumerator<ITask> BehaviorLoop()
{
   bool driveCommandFailed = false;
   ModeType previousMode = ModeType.NotSpecified;
   while (true)
   {
      driveCommandFailed = false;

      if (previousMode != _state.CurrentMode)
      {
         // update do UI
         WinFormsServicePort.FormInvoke(
            delegate()
            {
               _magellanUI.SetCurrentState(_state.CurrentMode.ToString());
            }
         );
      }
      previousMode = _state.CurrentMode;
```

```
            switch (_state.CurrentMode)
            {
               case ModeType.NotSpecified:
               case ModeType.Pronto:
                  yield return Arbiter.Receive(false, TimeoutPort(100), delegate(DateTime
timeout) { });
                  break;

               case ModeType.Reset:
                  {
                     if (!_iteratorsStarted)
                     {
                        // começa a processar as frames da câmara
                        SpawnIterator<DateTime>(DateTime.Now, ProcessFrame);

                        // começa a mostrar o tempo decorrido
                        SpawnIterator<DateTime>(DateTime.Now, UpdateElapsedTime);

                        // acede ao serviço sensor IR
                        _irSensorPort.Subscribe(_irNotifyPort);

                        // acede ao serviço que manipula as mensagens
                        MainPortInterleave.CombineWith(new Interleave(
                           new TeardownReceiverGroup(),
                           new ExclusiveReceiverGroup(),
                           new ConcurrentReceiverGroup
                           (
                              Arbiter.Receive<irsensor.Replace>(true, _irNotifyPort,
IRNotifyReplaceHandler)
                           )
                        ));

                        // evita a começo de iterações adicionais quando se reinicia o cenário
```

```
                _iteratorsStarted = true;
            }

            yield return Arbiter.Choice(_refereePort.Get(),
                delegate(referee.MagellanRefereeState state) { SetWaypoints(state); },
                delegate(Fault f) { }
                );

            _drivePort.EnableDrive(true);
            _drivePort.SetDriveSpeed(0, 0);
            _quadDrivePort.SetPose(new quadDrive.SetPoseRequestType(new
Microsoft.Robotics.PhysicalModel.Proxy.Pose()));
            _state.CurrentMode = ModeType.Pronto;

            break;
        }
    case ModeType.Procura:
        {
            // verifica se foi identificado um cone
            if (_imageProcessResult != null)
            {
                if (_imageProcessResult.Area > _imageProcessResult.AreaThreshold)
                {
                    // quando existe um cone na mira
                    _state.CurrentMode = ModeType.Aproximacao;
                    continue;
                }
            }

            quadDrive.DriveDifferentialFourWheelState quadDriveState = null;
            yield return Arbiter.Choice(_quadDrivePort.Get(),
                delegate(quadDrive.DriveDifferentialFourWheelState state)
```

```
        {
            quadDriveState = state;
        },
        delegate(Fault f) { }
    );

    if (quadDriveState == null)
        continue;

    double distance;
    double degrees;
    float currentHappiness = GetHappiness(quadDriveState.Position);
    if (currentHappiness > _previousHappiness)
    {
        // continua na mesma direcção
        distance = 0.25;
        degrees = 0;
    }
    else
    {
        // volta atrás e tenta de novo
        if(!driveCommandFailed)
            yield return Arbiter.Choice(
                _drivePort.DriveDistance(-0.25, _driveSpeed),
                delegate(DefaultUpdateResponseType response) { },
                delegate(Fault f) { driveCommandFailed = true; }
            );

        distance = 0.25;
        // escolhe um anglo entre -45 e -90
        degrees = (float)_random.NextDouble() * -45f - 45f;
        // reinicia o nivel de contentamento para a posição anterior
        _previousHappiness = _previousPreviousHappiness;
```

```
            }
            if (degrees != 0)
            {
                if (!driveCommandFailed)
                    yield return Arbiter.Choice(
                        _drivePort.RotateDegrees(degrees, _rotateSpeed),
                        delegate(DefaultUpdateResponseType response) { },
                        delegate(Fault f) { driveCommandFailed = true; }
                    );
            }

            if (!driveCommandFailed)
                yield return Arbiter.Choice(
                    _drivePort.DriveDistance(distance, _driveSpeed),
                    delegate(DefaultUpdateResponseType response) { },
                    delegate(Fault f) { driveCommandFailed = true; }
                );

            _previousPreviousHappiness = _previousHappiness;
            _previousHappiness = currentHappiness;

            break;
        }
    case ModeType.Aproximacao:
        {
            float IRDistance = -1f;

            // verifica a leitura do sensor IR
            irsensor.Get tmp = new irsensor.Get();
            _irSensorPort.Post(tmp);
            yield return Arbiter.Choice(tmp.ResponsePort,
                delegate(irsensor.AnalogSensorState state)
                {
```

```
            if (state.NormalizedMeasurement < 1f)
                IRDistance = (float)state.RawMeasurement;
        },
        delegate(Fault f) { }
        );

    if (IRDistance >= 0)
    {
        // confia no sensor IR para a aproximação final
        _state.CurrentMode = ModeType.AproximacaoFinal;
        break;
    }

    if ((_imageProcessResult == null) ||
        (_imageProcessResult.Area < _imageProcessResult.AreaThreshold))
    {
        // se perde cone, volta ao modo procura
        _state.CurrentMode = ModeType.Procura;
        continue;
    }
    float angle = _imageProcessResult.RightFromCenter * -2f * 5f;
    float distance = 0.25f;
    if (!driveCommandFailed)
        yield return Arbiter.Choice(
            _drivePort.RotateDegrees(angle, _rotateSpeed),
            delegate(DefaultUpdateResponseType response) { },
            delegate(Fault f) { driveCommandFailed = true; }
        );

    if (!driveCommandFailed)
        yield return Arbiter.Choice(
            _drivePort.DriveDistance(distance, _driveSpeed),
            delegate(DefaultUpdateResponseType response) { },
```

```
                delegate(Fault f) { driveCommandFailed = true; }
            );
        break;
    }
case ModeType.AproximacaoFinal:
    {
        // confia no sensor IR para a aproximação do cone
        float IRDistance = -1f;

        // leitura do sensor IR
        irsensor.Get tmp = new irsensor.Get();
        _irSensorPort.Post(tmp);
        yield return Arbiter.Choice(tmp.ResponsePort,
            delegate(irsensor.AnalogSensorState state)
            {
                if (state.NormalizedMeasurement < 1f)
                    IRDistance = (float)state.RawMeasurement;
            },
            delegate(Fault f) { }
            );

        if (IRDistance < 0)
        {
            // volta para a procura visual
            _state.CurrentMode = ModeType.Procura;
            break;
        }
        if (!driveCommandFailed)
            yield return Arbiter.Choice(
                _drivePort.DriveDistance(IRDistance, 0.2),
                delegate(DefaultUpdateResponseType response) { },
                delegate(Fault f) { driveCommandFailed = true; }
            );
```

```
            // marca o cone como visitado
            quadDrive.DriveDifferentialFourWheelState quadDriveState = null;
            yield return Arbiter.Choice(_quadDrivePort.Get(),
               delegate(quadDrive.DriveDifferentialFourWheelState state)
               {
                  quadDriveState = state;
               },
               delegate(Fault f) { }
            );

            if (quadDriveState != null)
            {
               for (int i = 0; i < _waypoints.Count; i++)
               {
                  float dx = (quadDriveState.Position.X - _waypoints[i].Location.X);
                  float dz = (quadDriveState.Position.Z - _waypoints[i].Location.Z);
                  float distance = (float)Math.Sqrt(dx * dx + dz * dz);
                  if (distance < 1f)
                     _waypoints[i].Visited = true;
               }
            }

            _state.CurrentMode = ModeType.Afasta;

            break;
         }

      case ModeType.Afasta:
         {
            // volta atrás e dá a volta
            if (!driveCommandFailed)
               yield return Arbiter.Choice(
```

```
                _drivePort.DriveDistance(-1, _driveSpeed),
                delegate(DefaultUpdateResponseType response) { },
                delegate(Fault f) { driveCommandFailed = true; }
            );

        if (!driveCommandFailed)
            yield return Arbiter.Choice(
                _drivePort.RotateDegrees(45, _rotateSpeed),
                delegate(DefaultUpdateResponseType response) { },
                delegate(Fault f) { driveCommandFailed = true; }
            );

        _state.CurrentMode = ModeType.Concluido;
        for (int i = 0; i < _waypoints.Count; i++)
        {
            if (!_waypoints[i].Visited)
            {
                _state.CurrentMode = ModeType.Procura;
                _previousHappiness = 0;
                _previousPreviousHappiness = 0;
                break;
            }
        }

        if (_state.CurrentMode == ModeType.Concluido)
        {
            // todos os cones foram visitados
            _endTime = DateTime.Now;
        }

        break;
    }
```

```
case ModeType.EvitaColisao:
    {
        // volta para trás
        if (!driveCommandFailed)
            yield return Arbiter.Choice(
                _drivePort.DriveDistance(-0.25, _driveSpeed),
                delegate(DefaultUpdateResponseType response) { },
                delegate(Fault f) { driveCommandFailed = true; }
            );

        // vira à esquerda
        if (!driveCommandFailed)
            yield return Arbiter.Choice(
                _drivePort.RotateDegrees(90, _rotateSpeed),
                delegate(DefaultUpdateResponseType response) { },
                delegate(Fault f) { driveCommandFailed = true; }
            );

        _state.CurrentMode = ModeType.Procura;

        // movimenta-se para a frente
        if (!driveCommandFailed)
            yield return Arbiter.Choice(
                _drivePort.DriveDistance(1, _driveSpeed),
                delegate(DefaultUpdateResponseType response) { },
                delegate(Fault f) { driveCommandFailed = true; }
            );
        break;
    }

case ModeType.Concluido:
    if (!driveCommandFailed)
        yield return Arbiter.Choice(
```

```
                    _drivePort.RotateDegrees(180, _rotateSpeed),
                    delegate(DefaultUpdateResponseType response) { },
                    delegate(Fault f) { driveCommandFailed = true; }
                );
            break;

        }
    }
}

private void SetWaypoints(referee.MagellanRefereeState state)
{
    foreach (Vector3 location in state.Cones)
        _waypoints.Add(new Waypoint(location));
}

private IEnumerator<ITask> UpdateElapsedTime(DateTime timeout)
{
    string previous = string.Empty;
    while (true)
    {
        string newString = string.Empty;

        switch (_state.CurrentMode)
        {
            case ModeType.NotSpecified:
            case ModeType.Pronto:
            case ModeType.Reset:
                newString = string.Format("Elapsed: {0}:{1:D2}", 0, 0);
                break;

            case ModeType.Procura:
            case ModeType.Aproximacao:
```

```
                case ModeType.AproximacaoFinal:
                case ModeType.Afasta:
                    {
                        TimeSpan elapsed = DateTime.Now - _startTime;
                        newString = string.Format("Elapsed: {0}:{1:D2}", elapsed.Minutes,
elapsed.Seconds);
                        break;
                    }

                case ModeType.Concluido:
                    {
                        TimeSpan elapsed = _endTime - _startTime;
                        newString = string.Format("Elapsed: {0}:{1:D2}", elapsed.Minutes,
elapsed.Seconds);
                        break;
                    }
            }

            if (newString != previous)
            {
                previous = newString;
                // faz o update do UI
                WinFormsServicePort.FormInvoke(
                    delegate()
                    {
                        _magellanUI.SetElapsedTime(newString);
                    }
                );
            }

            yield return Arbiter.Receive(false, TimeoutPort(100), delegate { });
        }
    }
```

```
private IEnumerator<ITask> ProcessFrame(DateTime timeout)
{
   while (true)
   {

      yield return Arbiter.Choice(
         _cameraPort.QueryFrame(),
         ValidateFrameHandler,
         DefaultFaultHandler);

      yield return Arbiter.Receive(false, TimeoutPort(200), delegate { });
   }
}

private void ValidateFrameHandler(webcam.QueryFrameResponse cameraFrame)
{
   try
   {
      if (cameraFrame.Frame != null)
      {
         DateTime begin = DateTime.Now;
         double msFrame = begin.Subtract(cameraFrame.TimeStamp).TotalMilliseconds;

         // Ignora imagens antigas
         if (msFrame < 1000.0)
         {
            _imageProcessResult = ProcessImage(cameraFrame.Size.Width,
cameraFrame.Size.Height, cameraFrame.Frame);
            double msProcessing = DateTime.Now.Subtract(begin).TotalMilliseconds;
            LogVerbose(LogGroups.Console, string.Format("{0} Frame {1} ms Processed
{2} ms", begin, msFrame, msProcessing));
```

```
                WinFormsServicePort.FormInvoke(
                    delegate()
                    {
                        _magellanUI.SetCameraImage(cameraFrame.Frame, _imageProcessResult);
                    }
                );
            }
            else
            {
                LogVerbose(LogGroups.Console, string.Format("Frame discarded {0} ms old",
msFrame));
            }
        }
    }
    catch (Exception ex)
    {
        LogError(ex);
    }
}

private ImageProcessResult ProcessImage(int width, int height, byte[] pixels)
{
    if (pixels == null || width < 1 || height < 1 || pixels.Length < 1)
        return null;

    int offset = 0;
    float threshold = 2.5f;
#if false
    int[] xProjection = new int[width];
    int[] yProjection = new int[height];
#endif
    int xMean = 0;
    int yMean = 0;
```

```
        int area = 0;

        // apenas processa cada quarto pixel
        for (int y = 0; y < height; y += 2)
        {
          offset = y * width * 3;
          for (int x = 0; x < width; x += 2)
          {
            int r, g, b;

            b = pixels[offset++];
            g = pixels[offset++];
            r = pixels[offset++];

            float compare = b * threshold;
            if((g > compare) && (r > compare))
            {
              // a côr é amarelada
              xMean += x;
              yMean += y;
              area++;

              // decomposição da côr
              pixels[offset - 3] = 255;
              pixels[offset - 2] = 0;
              pixels[offset - 1] = 255;
#if false
              xProjection[x]++;
              yProjection[y]++;
#endif
            }
            offset += 3;    // salta um pixel
          }
```

```
        }

        if (area > 0)
        {
            xMean = xMean / area;
            yMean = yMean / area;
            area *= 4;
        }

        ImageProcessResult result = new ImageProcessResult();

        result.XMean = xMean;
        result.YMean = yMean;
        result.Area = area;
        result.RightFromCenter = (float)(xMean - (width / 2)) / (float)width;
        result.DownFromCenter = (float)(yMean - (height / 2)) / (float)height;

#if false
        int xOff = -xMean;
        int xSecond = 0;
        int xThird = 0;

        for (int i = 0; i < xProjection.Length; i++)
        {
            if (xProjection[i] > 0)
            {
                int square = xOff * xOff * xProjection[i];

                xSecond += square;
                xThird += xOff * square;
            }
            xOff++;
        }
```

```
            double xStdDev = Math.Sqrt((double)xSecond / area);
            double xSkew = xThird / (area * xStdDev * xStdDev * xStdDev);

            int yOff = -yMean;
            int ySecond = 0;
            int yThird = 0;

            for (int i = 0; i < yProjection.Length; i++)
            {
               if (yProjection[i] > 0)
               {
                  int square = yOff * yOff * yProjection[i];

                  ySecond += square;
                  yThird += yOff * square;
               }
               yOff++;
            }

            double yStdDev = Math.Sqrt((double)ySecond / area);
            double ySkew = yThird / (area * yStdDev * yStdDev * yStdDev);

            result.XStdDev = xStdDev;
            result.YStdDev = yStdDev;
            result.XSkew = xSkew;
            result.YSkew = ySkew;
#endif
            return result;
        }


        // Manuseamento base de erros
```

```
void DefaultFaultHandler(Fault fault)
{
   LogError(fault);
}

void IRNotifyReplaceHandler(irsensor.Replace replace)
{
   if (replace.Body.RawMeasurement < 1f)    // perto de um metro de distancia
   {
      if (_state.CurrentMode == ModeType.Procura)
      {

         _drivePort.AllStop();
         _state.CurrentMode = ModeType.EvitaColisao;
      }
   }
}

float GetHappiness(Vector3 pos)
{
   float happiness = 0;
   for(int i=0; i<_waypoints.Count; i++)
   {
      if (_waypoints[i].Visited)
         continue;

      float dx = (pos.X - _waypoints[i].Location.X);
      float dz = (pos.Z - _waypoints[i].Location.Z);
      float distance = (float)Math.Sqrt(dx * dx + dz * dz);

      float proximity = 50f - Math.Min(distance, 50f);
      if(proximity > 40f)
```

```
                proximity = proximity * proximity;

                happiness += proximity;
            }
            return happiness;
        }
    }

    class Waypoint
    {
        public Vector3 Location;
        public bool Visited;

        public Waypoint(Vector3 location)
        {
            Location = location;
            Visited = false;
        }
    }
}
```

Anexo 2 – Especificações para executar a simulação

Para executar esta simulação é necessário possuir instalado o Microsoft Robotics Studio 1.5 e o Visual Studio 2005.

No cd disponibilizado basta copiar a pasta referente ao projecto que se chama apps que está presente na pasta projecto para o directório principal onde está instalado o Microsoft Robotics Studio que por definição é C:\Microsoft Robotics Studio (1.5)\.

É necessário também copiar os ficheiros que se encontram na pasta dll presente no cd dentro da pasta projecto para a pasta C:\Microsoft Robotics Studio (1.5)\bin\.

Em seguida basta abrir a solução presente na pasta apps copiada para o directório do Microsoft Robotics e já no Visual Studio 2005 fazer o build da solução e depois de concluído fazer debug e a simulação arrancará.

Quando a simulação se encontra a correr, terá ao dispor uma consola com 2 botões, o botão de start e reset. Sempre que a simulação é iniciada para que o robot fique pronto a realizar a sua tarefa, terá que pressionar o botão reset, e para que ele realize a sua tarefa terá que pressionar o botão start. Sempre que o robot se encontrar a realizar a tarefa pode sempre reiniciar a simulação quando quiser pressionando o botão reset. O robot dá por concluída a sua tarefa quando se encontra em rotação de 180 graus sobre si próprio.

Anexo 3 – Documento de requisitos

Descrição Geral do Trabalho

Para este trabalho decidimos apresentar um cenário de Vigilância de Áreas Urbanas, através da Vigilância Activa em Robótica Móvel utilizando o Microsoft Robotics.

Vamos então considerar para este trabalho agentes robóticos, com lagartas para melhor locomoção e para melhor capacidade de ultrapassagem de obstáculos. Terão também um sistema integrado de GPS para conhecimento da sua posição e para poder também se deslocar para novas posições.

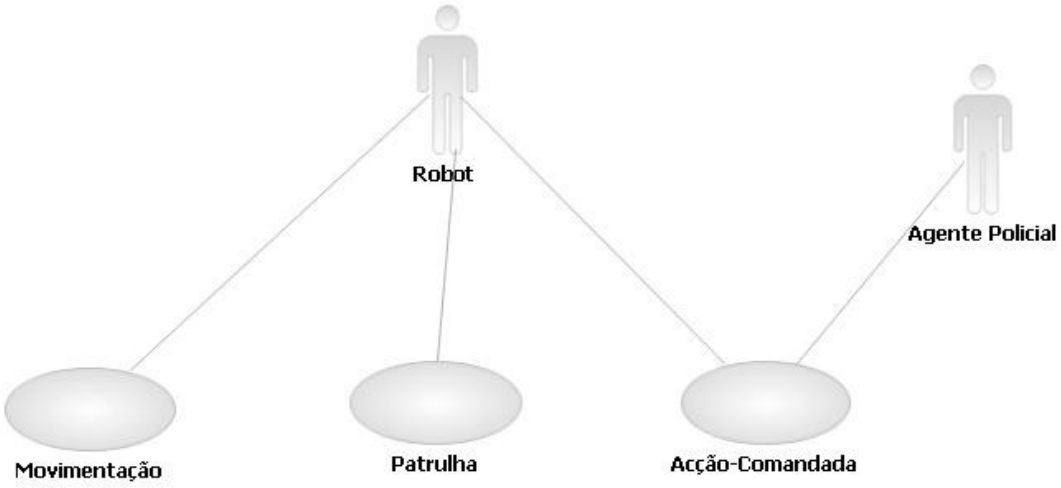
Estarão também munidos de sensores e câmaras, para ultrapassar obstáculos e identificação dos mesmos, mas também para avaliação de ameaças. Para controlo de ameaças os agentes robóticos poderão possuir armas, focos de luz, câmaras nocturnas, dispositivos de sons, etc.

O objectivo destes agentes robóticos será a identificação, controlo e regulação de possíveis ameaças numa área urbana específica, podendo ser operados também por agentes policiais em certas tarefas.

Casos de Uso

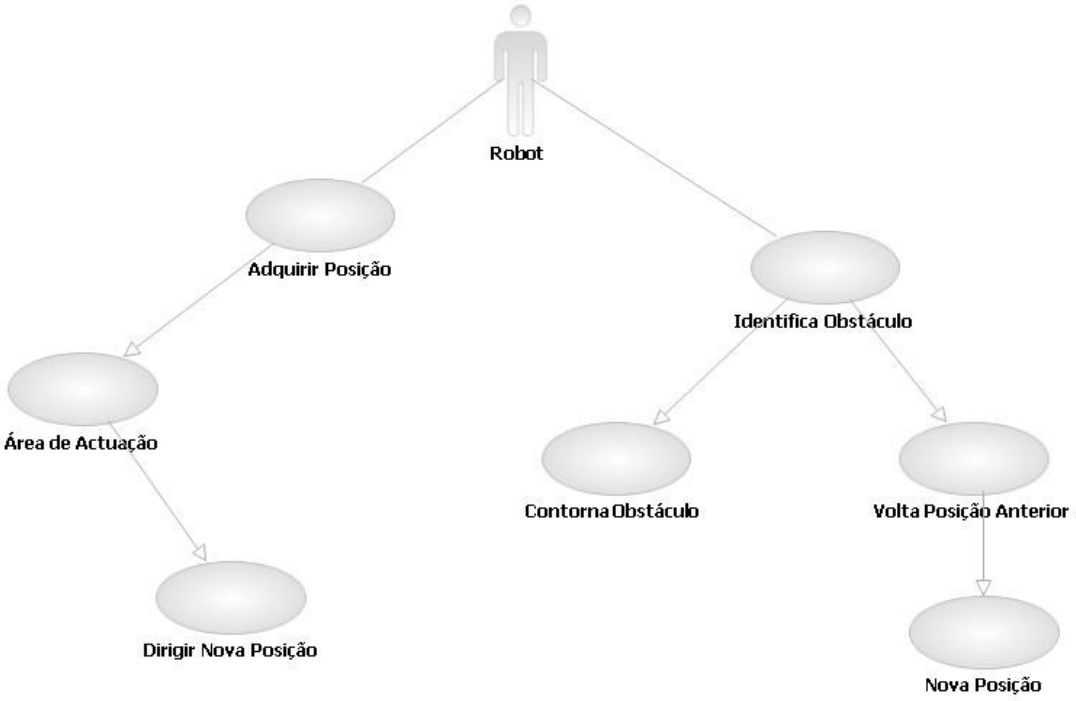
Geral

Foram identificadas por nós 3 tarefas essenciais para a funcionalidade do mesmo. Existem dois actores sendo eles o próprio agente robótico e agentes policiais.



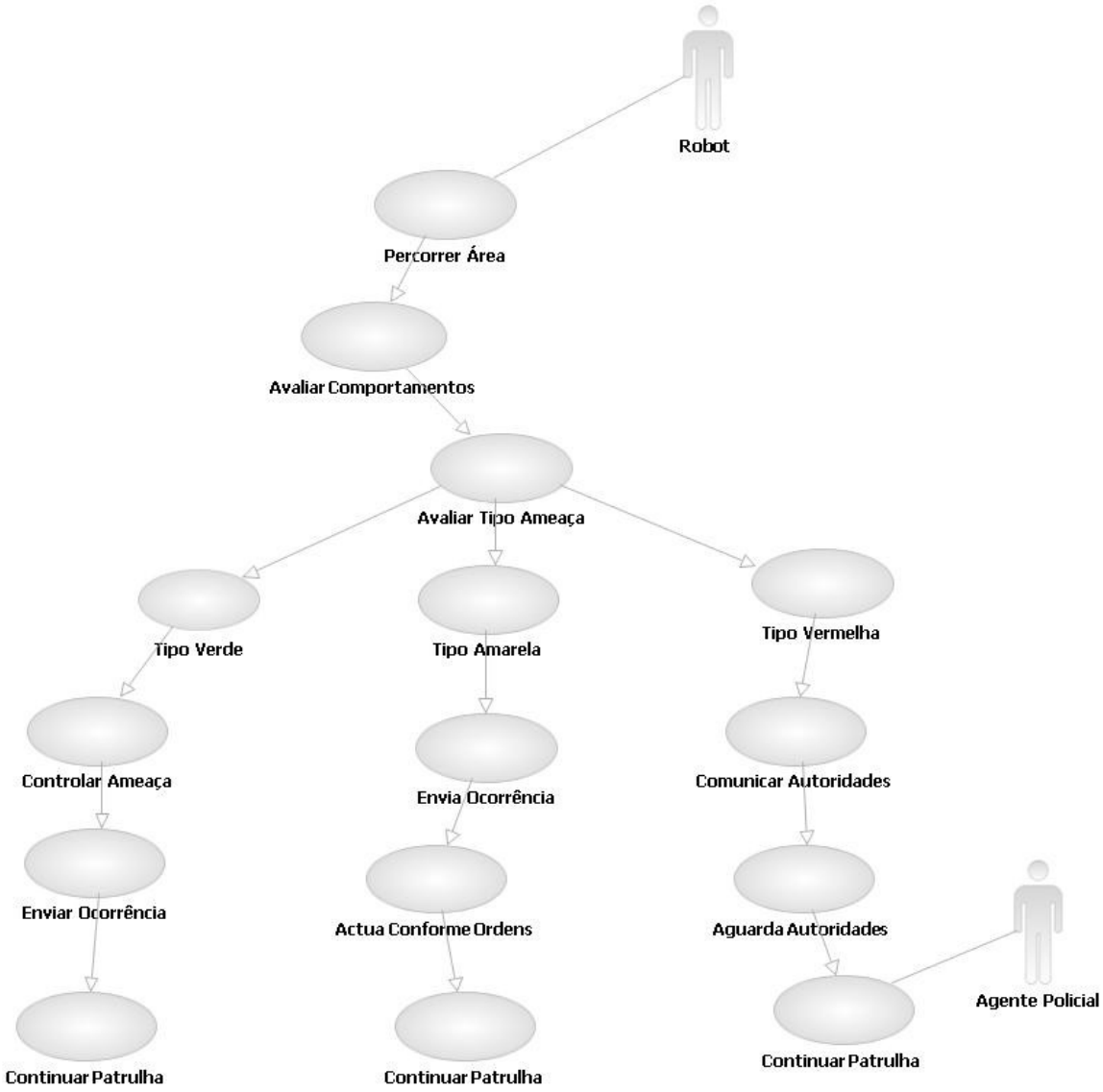
Movimentação

Aqui teremos os casos de uso referentes à movimentação do agente robótico, terá uma área configurada previamente por coordenadas GPS e apenas operará nessa área, tendo que contornar obstáculos e não poderá alterar o comportamento normal da sociedade habitante dessa mesma área.



Patrulha

O agente robótico terá que ser capaz de "conviver" com as pessoas no dia-a-dia, percorrendo a área a que está destinado, avaliando comportamentos das mesmas e identificar possíveis ameaças atribuindo-lhes um grau conforme a gravidade da situação. Terá que ser capaz também de controlar algumas delas recorrendo a dispositivos instalados no agente. O agente robótico terá de enviar as ocorrências para o centro de controlo.



Acção Comandada

Este agente robótico terá a capacidade de em certas intervenções policiais, mesmo algumas detectadas por eles próprios, de serem comandados remotamente por agentes policiais, para efectuar certas intervenções em ameaças que a presença policial será obrigatória.