



EPS

Escola Politècnica
Superior

Projecte/Treball Fi de Carrera

Estudi: Eng. Tècn. Informàtica de Sistemes. Pla 2001

Títol: Programació del robot e-puck amb Microsoft Robotics Studio.

Document: Memòria

Alumne: Alex García Agustí

Director/Tutor: Marc Carreras Pèrez

Departament: Arquitectura i Tecnologia de Computadors

Àrea: ATC

Convocatòria (mes/any): 09/09

INDEX

1. Introducció	3
1.1. Antecedents.....	3
1.2. Motivació i abast.....	3
1.3. Objectius	3
1.4. Planificació.....	4
1.5. Calendari del Projecte.....	5
1.6. Estructura de la memòria.....	6
2. Eines de desenvolupament	7
2.1. Microsoft Robotics Studio.....	7
2.1.1. Introducció	7
2.1.2. Simular aplicacions en 3D.....	7
2.1.3. Programació visual.....	8
2.1.4. Interacció amb robots mitjançant interfície basada en web.....	9
2.1.5. Arquitectura de Microsoft Robotics Studio.....	10
2.1.6. Reutilitzar serveis modulars.....	11
2.1.7. Llenguatges que es poden utilitzar en MRS.....	13
2.1.8. Versions.....	13
2.1.9. Llicència.....	13
2.2. Microsoft Visual C#.....	15
2.2.1. Introducció..	15
2.2.2. Principals característiques.....	15
2.3. Robot E-puck	17
2.3.1. Descripció general.....	17
2.3.2. Sensors.....	19
2.3.3. Actuadors.....	22
2.3.4. Connectivitat i comunicació.....	24
3. Simulació del robot e-puck	26
3.1. Introducció.....	26
3.2. Implementació	26
3.2.1. Crear el projecte.....	26
3.2.2. Afegir components.....	26
3.2.3. Creació d'una classe identitat per al robot e-puck.....	28
3.2.4. Creació d'una malla.....	30
3.2.5. Webcam.....	31
3.2.6. Sensors IR.....	31
3.2.7. Modificació del manifest.....	33
4. Programa de control automàtic sobre la simulació.	35
4.1. Introducció.....	35
4.2. Implementació.....	36
4.2.1. Components	36
4.2.2. Definir els estats.....	37
4.2.3. Definir els Handlers.....	38
4.2.4. Mètodes auxiliars per el servei Drive.....	41
4.2.5. Ignorar antigues notificacions.....	42
4.2.6. Modificació del mètode Start().....	43
4.2.7. Modificació del manifest.....	44
5. Interfície de connexió i comunicació entre el robot e-puck i MRS	45
5.1. Introducció.....	45
5.2. Implementació	46
5.2.1. Afegir llibreries necessàries i instància del port.....	46
5.2.2. Connexió amb el robot.....	46

5.2.3. Mètodes Set	46
5.2.4. Mètodes Get	47
5.2.5. Altres mètodes.....	48
5.2.6. Mètodes auxiliars del port.....	48
5.2.7. Desconnexió del robot.....	48
5.3. Utilització de la classe.....	48
6. Aplicació gràfica de monitorització i control del e-puck.....	49
6.1. Introducció.....	49
6.2. Disseny i Implementació.....	49
6.2.1. Crear el projecte i agregar els elements necessaris.....	49
6.2.2. Creació dels botons de control de motors.....	50
6.2.3. Implementació dels mètodes de control dels motors.....	50
6.2.4. Creació dels botons de control de tots els leds.....	51
6.2.5. Implementació dels mètodes de control dels leds.....	51
6.2.6. Disseny dels elements d'obtenció de dades.....	52
6.2.7. Implementació dels mètodes d'obtenció de dades.....	53
6.2.8. Disseny dels elements necessaris per reproduir un so.....	54
6.2.9. Implementació del mètode per reproduir un so a l'altaveu.....	54
6.2.10. Creació i implementació del botó Borrar.....	54
6.2.11. Implementació d'una petita rutina.....	55
6.2.12. Disseny dels elements de connexió.....	56
6.2.13. Implementació del mètode de connexió.....	56
7. Resultats del programa de control automàtic sobre el robot real.....	58
7.1. Introducció.....	58
7.2. Implementació.....	58
7.2.1. Crear el projecte i agregar els elements necessaris.....	58
7.2.2. Implementació del mètode Start().....	59
7.2.3. Implementació del mètode per obtenir la distància.....	60
7.2.4. Elecció de les distàncies de seguretat i d'aturada.....	61
7.3. Resultats i conclusions.....	61
8. Conclusions i treballs futurs.....	63
8.1. Valoració global	63
8.2. Assoliment d'objectius.....	64
8.3. Treballs futurs	65
9. Bibliografia.....	66
Annex A: Manual d'usuari.....	67
A.1. Instal·lació Microsoft Robotics Studio.....	67
A.2. Instal·lació de Microsoft Visual C#.....	68
A.3. Còpia dels projectes.....	68
A.4. Configurar Bluetooth	69
A.5. Execució dels programes.....	70

1.INTRODUCCIÓ

1.1. Antecedents

Al laboratori docent de robòtica s'utilitzen robots mòbils autònoms per treballar aspectes relacionats amb el posicionament, el control de trajectòries, la construcció de mapes... Es disposa de cinc robots comercials anomenats "e-puck" que es caracteritzen per les seves dimensions reduïdes, i un conjunt complet de sensors i actuadors. Es disposa també d'un entorn de proves on els robots es poden moure i evitar obstacles.

1.2. Motivació

En el laboratori de robòtica es vol treballar amb el entorn de control i simulació de robots Microsoft Robotics Studio (MRS).

Microsoft Robotics Studio (MRS) és un entorn per a crear aplicacions per a robots utilitzant una gran varietat de plataformes hardware. Conté un entorn de simulació en el que es pot modelar i simular el moviment del robot. Permet també programar el robot, i executar-lo en l'entorn simulat o bé en el real. MRS resol la comunicació entre els diferents processos asíncrons que solen estar presents en el software de control d'un robot: processos per atendre sensors, actuadors, sistemes de control, sistemes de percepció, comunicacions amb l'exterior,... MRS es pot utilitzar per modelar nous robots utilitzant components que ja estiguin disponibles en les seves llibreries, o també permet crear component nous.

Per tal de conèixer en detall aquesta eina, es va decidir modelar els robots e-pucks, simular-los, realitzar un programa de control, realitzar la interfície amb el robot i comprovar el funcionament amb el robot real. El resultat del projecte permetrà a futurs estudiants aprendre i treballar amb els robots e-puck i la MRS.

1.3 Objectius

Els objectius d'aquest projecte són els següents:

- 1 Estudi de Microsoft Robotics Studio: aprendre a utilitzar els seus components i entendre el funcionament intern.
- 2 Modelar i simular el robot e-puck amb els seus actuadors (2 motors) i sensors bàsics (sensors d'infrarojos).
- 3 Realitzar un programa que controli el robot de forma autònoma en el simulador i comprovar-ne el seu funcionament. Aquest programa ha de moure el robot sense topar amb els objectes que trobi.
- 4 Dissenyar i implementar la interfície de comunicació entre el robot real i MRS.
- 5 Realitzar proves amb el robot real, analitzar i valorar el funcionament de MRS.

1.4 Planificació

Per planificar aquest projecte s'han fixat els objectius principals de forma ordenada segons si eren necessaris o preferiblement tenir assolits abans d'abordar el següent objectiu, llavors s'han pensat quines son les tasques necessàries que s'han de complir per tal d'assolir cada objectiu i s'ha assignat un temps orientatiu per a cadascuna.

Tot seguit s'expliquen les tasques que s'han dut a terme en cada part.

1. Estudi de Microsoft Robotics Studio:

Per poder començar a realitzar el projecte primer va ser necessari un aprenentatge previ sobre aquest entorn per poder comprendre el seu funcionament i les seves possibilitats.

Per tant la primera tasca consistia en investigar què és aquesta plataforma buscant en la seva pagina web, fòrums, documentació, etc.

La segona tasca consistia en descarregar la plataforma i instal·lar-la, i realitzar uns tutorials recomanats per els mateixos creadors d'aquesta plataforma.

Un cop acabades aquestes tasques es pot començar amb el següent pas que varem decidir que seria la part de simulació del robot e-puck utilitzant les eines del MRS.

2. Modelar i simular un e-puck bàsic:

Per poder simular el robot e-puck primerament s'havia d'investigar com s'havia fet per modelar altres robots que ja han estat modelats per els creadors del MRS.

En acabat el següent pas va ésser crear un nou projecte que introdueix els elements necessaris en una simulació i tot seguit hi afegís el robot.

Per el robot primer s'ha de crear l'estructura i després afegir-hi les rodes, els motors, i els sensors bàsics.

L'última tasca consistia en crear una imatge per el robot per tal de que la simulació fos el més real possible.

3. Realitzar un programa que controli el robot simulat.

Per poder realitzar aquest programa primerament s'havia d'investigar com comunicar un programa de control amb una simulació. Per fer això s'havia d'observar com es feia en altres robots.

En acabat es va decidir què havia de fer el robot en aquest programa i es va fer el disseny d'aquest programa de forma descendent definint les funcions necessàries, tot això en pseudocodi.

El següent pas va ser escriure el codi i les funcions pròpies del MRS necessàries perquè el programa pugui funcionar correctament.

L'última tasca consisteix en realitzar les proves necessàries per assegurar el bon funcionament del programa i fer les necessàries correccions.

4. Crear una interfície de comunicació amb el robot real.

Per realitzar aquesta interfície primer va ser necessari investigar com comunicar-se amb el robot mitjançant el port bluetooth, és a dir, quin protocol utilitza el robot e-puck.

Un cop comprès el protocol es va decidir que la millor forma de comunicar-se era utilitzant una única classe on s'implementin mètodes per comunicar-se amb el robot utilitzant un port sèrie associat al bluetooth del propi ordinador.

Després de dissenyar aquesta classe s'ha d'implementar en el llenguatge C#.

5. Crear una aplicació gràfica de monitorització i control del e-puck

Per realitzar aquesta aplicació primer es van estudiar els elements de Visual C# que permeten crear aquestes aplicacions i tots els seus components.

Després de comprendre com funcionen aquests elements es va crear el projecte i es van començar a dissenyar els botons i quadres de text necessaris per controlar els actuadors de l'e-puck, i obtenir i mostrar els valors que retornen els sensors. A continuació es van crear aquests botons i quadres de text i es van ordenar segons el disseny previ.

Tot seguit es van implementar els mètodes que s'executen al fer clic sobre els botons per tal de que el robot executi les comandes enviades, respongui a les peticions de l'estat dels sensors i es mostrin els resultats.

Finalment es van realitzar proves per comprovar el funcionament de l'aplicació.

6. Controlar el robot real amb el mateix algorisme que el robot simulat

Per controlar el robot real amb el mateix algorisme primerament s'ha dissenyat l'algorisme utilitzant els mètodes de la interfície de comunicació.

A continuació s'ha creat un nou projecte i s'ha implementat l'algorisme en llenguatge C#. Després s'han adaptat els elements necessaris perquè es pugui executar l'algorisme correctament en el robot real.

Finalment s'han realitzat proves per comprovar el funcionament de l'algorisme i treure'n conclusions.

1.5. Calendari del Projecte

La realització del projecte s'ha estructurat de la següent manera tal com es mostra a la figura 1.1:

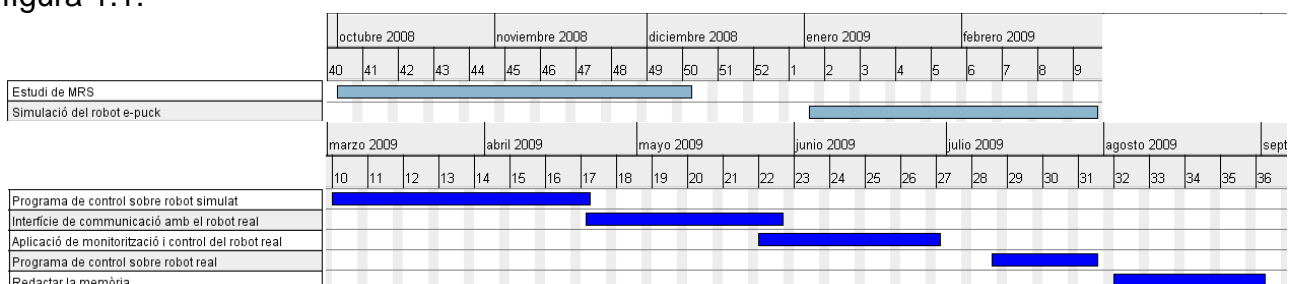


Figura 1. 1: Diagrama de Grantt del projecte

1.6. Estructura de la Memòria

L'estructura del projecte és la següent:

Al capítol 2, s'explica tot l'entorn de treball, les eines utilitzades i les seves característiques.

Al capítol 3, s'explica tota la part de simulació del robot e-puck i l'entorn de proves.

Al capítol 4 es descriu el disseny i la implementació d'un programa de control sobre la simulació.

Al capítol 5 es descriu el disseny i la implementació d'una interfície de comunicació amb el robot e-puck real.

Al capítol 6 es descriu la creació d'una aplicació que permet controlar el robot i obtenir els valors que recullen els sensors d'aquest.

Al capítol 7 es descriu l'adaptació del programa de control del capítol 4 per treballar sobre el robot real i les conclusions que s'han obtingut.

Finalment, al capítol 8 s'expliquen les conclusions i els treballs futurs.

L'Annex es un manual d'usuari on s'explica com instal·lar els programes necessaris i com executar les aplicacions que formen part del projecte.

2. EINES DE DESENVOLUPAMENT

2.1. Microsoft Robotics Studio

2.1.1. Introducció

Microsoft Robotics Studio és un entorn basat en Windows per a que desenvolupadors aficionats, acadèmics o comercials puguin crear aplicacions robòtiques per a gran varietat de plataformes de hardware. MRS té un entorn d'execució lleuger, inclou una rutina orientada a serveis, un conjunt de eines de simulació, així com tutorials i codi d'exemple per a iniciar-se.

És una plataforma de desenvolupament extrem a extrem, permet als programadors crear serveis per una gran varietat de hardware robòtic.

A continuació s'expliquen les característiques més importants de Microsoft Robotics Studio.

2.1.2. Simular aplicacions en 3D

Microsoft Robotics Studio pretén que la robòtica arribi a un rang ampli de gent i que acceleri el seu desenvolupament i adopció. Una part molt important en aquest esforç és la simulació avançada de la física funcionant en temps real (veure figura 2.1).



Figura 2.1: Simulacions 3D de múltiples robots

S'ha definit la rutina de simulació per a que es pugui utilitzar en un gran rang d'escenaris amb altes demandes de fidelitat, visualització i escala. Al mateix temps, un usuari novell sense nocions de programació pot utilitzar la simulació, programant aplicacions en un entorn similar als videojocs.

MRS permet simular aplicacions robòtiques en uns models realistes en 3D (figura 2.2) .

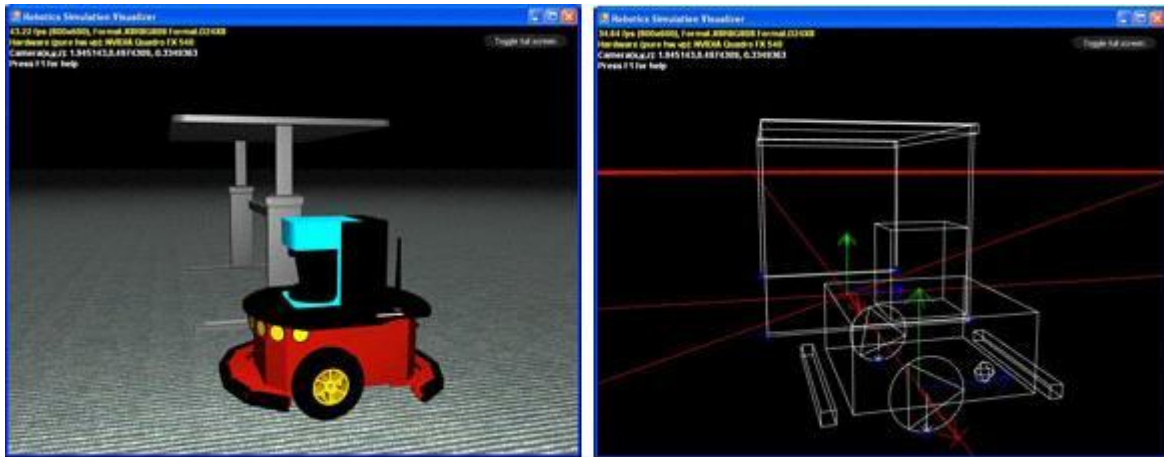


Figura 2.2: A l'esquerre visió de la simulació realista i a la dreta visió de la simulació física

La eina de simulació de MRS inclou *AGEIA™ PhysX™ Technology* de *AGEIA Technologies Inc.*, una empresa pionera en físiques accelerades per hardware, permetent crear simulacions en el món real per a models de robots. Les simulacions *PhysX* es poden accelerar amb el hardware *AGEIA*.

El motor de renderització està basat en *Microsoft XNA Framework*, utilitzada per al desenvolupament de videojocs per PC i Xbox360.

2.1.3. Programació visual

Una funció important de MRS és el *Visual Programming Language (VPL)*, que permet a qualsevol usuari crear i provar les aplicacions robòtiques molt fàcilment. És tan fàcil com arrossegat i deixar anar blocs que representen serveis i connectar-los entre ells, com es veu a la figura 2.3. També es poden utilitzar una col·lecció de blocs connectats com a un de sol i utilitzar-ho en qualsevol part del programa.

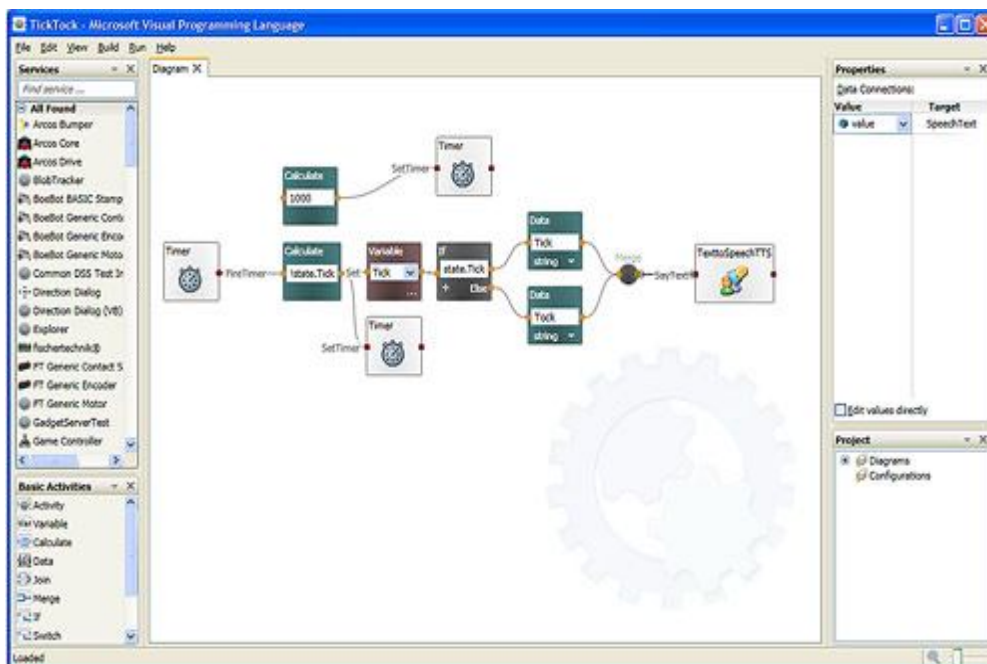


Figura 2.3: Interfície d'usuari de *Visual Programming Language*

VPL és un entorn de programació dissenyat per a una programació basada en flux de dades gràfic, en comptes del flux de control que trobem a la programació convencional. En comptes d'una sèrie de comandes imperatives executades seqüencialment, un programa en flux de dades és com una sèrie de treballadors a una línia de muntatge, on cadascun té la seva tasca quan el material arriba. Com a resultat VPL està dissenyat per a programar una varietat d'escenaris de processament concurrents o distribuïts.

L'objectiu de VPL és que els programadors inexperts entenguin fàcilment els conceptes com variables i lògica. De totes formes VPL no està limitat a novells, la natura del llenguatge de programació pot significar als programadors experts una forma ràpida de fer prototips o desenvolupar codi. Cal afegir que mentre el seu disseny superior és ideal per a programar aplicacions per a robots, l'arquitectura subjacent no està limitada a això. Com a resultat VPL té una gran audiència de públic de més àmbits. La programació en flux de dades de MRS consisteix en una seqüència d'activitats connectades, representades com a blocs amb entrades i sortides que es poden connectar a altres activitats.

2.1.4. Interacció amb robots mitjançant interfície basada en web

MRS permet crear aplicacions que proporcionen a l'usuari la possibilitat de fer un seguiment o controlar un robot remotament utilitzant un explorador web i transmetent comandes mitjançant les tecnologies web, com html o Javascript, així com muntar càmeres als robots i controlar-los en entorns remots (veure figura 2.4).

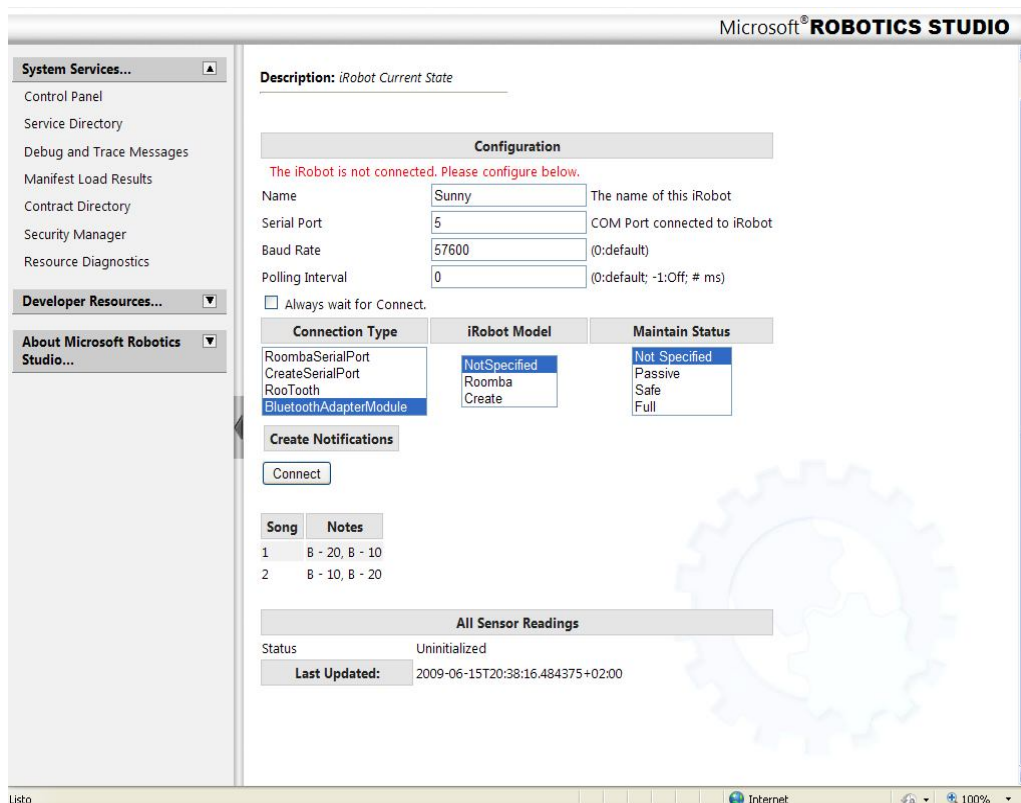


Figura 2.4: Accedint al robot utilitzant una interfície web

El desenvolupador pot accedir fàcilment als sensors i actuadors del robot, gràcies a una llibreria d'implementació de concurrència desenvolupada en .NET . La comunicació està basada en missatges, permetent la comunicació entre mòduls. Això és un entorn d'execució lleuger **orientat a serveis**.

2.1.5. Arquitectura de Microsoft Robotics Studio

La tasca principal de les aplicacions robòtiques és la de processar senyals d'entrada de sensors des d'una sèrie de fonts i organitzar una sèrie d'actuadors per a respondre a les entrades de forma que es segueixi el propòsit de l'aplicació.

La figura 2.5 mostra un diagrama de flux de dades d'una aplicació robòtica que conté un sensor (bumper) que avisa quan ha estat tocat, un actuador (message box) que controla la pantalla, i un orquestrador que connecta i organitza el sistema:

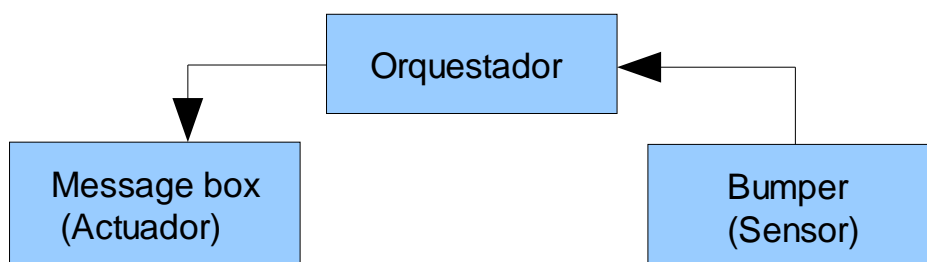


Figura 2.5: Diagrama de flux de dades d'una aplicació robòtica

La base de l'arquitectura de Microsoft Robotics Studio són les rutines: **Concurrència i Coordinació (CCR)** i **Serveis de Software Descentralitzat (DSS)**.

Concurrència i Coordinació (CCR)

La *Concurrency and Coordination Runtime* (Rutina de Concurrència i Coordinació - CCR) fa simple la programació asíncrona. Fa simple escriure programes per gestionar entrades asíncrones de múltiples sensors de robots i sortides a motors i actuadors.

CCR proveeix un model de programació molt concurrent, orientat a missatges amb unes primitives d'orquestració que permeten la coordinació de missatges sense haver-ho de fer de forma manual amb semàfors, *threads*, etc.

CCR direcciona la necessitat d'aplicacions orientades a servei proveint un model de programació que faciliti la gestió d'operacions asíncrones, interactuant amb la concurrència, aprofitant el hardware paral·lel i gestionant errors parcials. CCR és apropiat per a un model d'aplicació que separa components en trossos que es poden comunicar únicament mitjançant missatges.

Això permet a l'usuari dissenyar les seves aplicacions de forma que els seus mòduls de software o components es puguin connectar fàcilment. Es poden desenvolupar independentment sense haver de tenir en compte els altres components.

Aquest enfocament canvia la forma de pensar del programador des del principi del procés de disseny i facilita interactuar amb concurrència i errors de forma consistent.

Serveis de Software Descentralitzat (DSS)

El model d'aplicació de *Decentralized Software Services* (Serveis de Software Descentralitzats - DSS) fa simple accedir i respondre a l'estat d'un robot utilitzant un navegador web o una aplicació basada en Windows.

DSS proveeix un model d'aplicació lleuger, orientat a servei que combina termes de l'arquitectura basada en web tradicional (anomenada *REST*) amb part de l'arquitectura de serveis web (*Web Services architecture*). El model d'aplicació definit per DSS s'ha construït sobre el model *REST* controlant els serveis a través del seu estat juntament amb un conjunt d'operacions sobre aquest estat que amplien el model d'aplicació de *HTTP* afegint manipulació, estructura de dades, notificació de events i composició de serveis. L'objectiu principal de DSS és promoure la simplicitat, interoperabilitat i facilitat de connexió. Això ho fa ideal per a crear aplicacions com a composicions de serveis a pesar de que aquests serveis estiguin funcionant sense el mateix node o a través de xarxa. El resultat és una plataforma flexible però simple per escriure un ampli conjunt d'aplicacions.

La rutina DSS està implementada a més alt nivell que CCR i no depèn de cap altre component de Microsoft Robotics Studio. Proveeix un entorn d'emmagatzemament per a serveis de gestió i un conjunt de serveis d'infraestructura que es poden usar per a creació, descobriment, proves, supervisió i seguretat de serveis.

2.1.6. Reutilitzar serveis modulars

Un servei és el bloc bàsic per a construir aplicacions utilitzant Microsoft Robotics Studio i el seu component clau del model d'aplicació DSS. Els serveis es poden utilitzar per representar, per exemple:

- Components Hardware: sensors, actuadors
- Components Software: interfície d'usuari, emmagatzemament, etc
- Agregacions: sensors de fusió, etc.

Els serveis s'executen dins el context d'un node DSS. Un node DSS és un entorn hosting que proveeix suport als serveis per a ser creats o gestionats fins que s'eliminen o el node DSS para. Els serveis són inherents a la xarxa i es poden comunicar amb els demès d'una forma uniforme encara que no estiguin executant-se dins el mateix node DSS. El model de servei DSS s'ha dissenyat per a facilitar la reutilització de serveis fent-los fàcils d'utilitzar i de combinar entre ells.

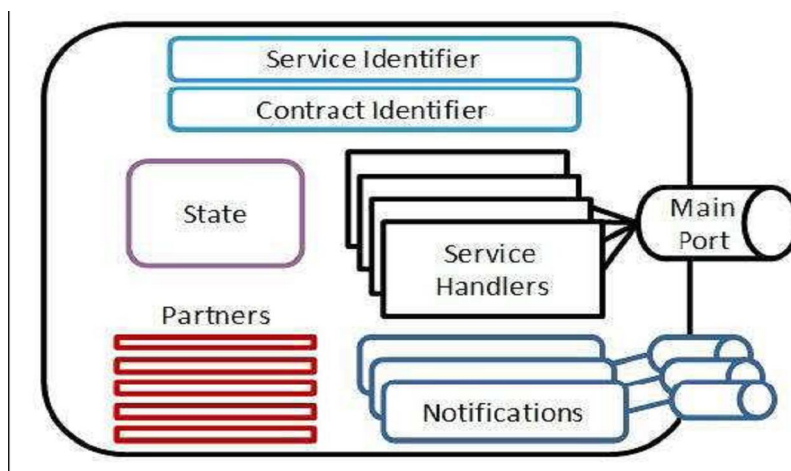


Figura 2.6: Model de serveis DSS

Tots els serveis DSS disposen d'una sèrie de components comuns(veure figura 2.6), els principals són:

- **Service Identifier (Identificador de Servei):** proporciona identitat al servei, donant lloc a que altres serveis puguin comunicar-se amb ell o permetent fins i tot al navegador web accedir-hi.
- **Contract Identifier (Identificador de Contracte):** identificació única de la funcionalitat que implementa el servei, de forma que altres serveis ho puguin reutilitzar.
- **Service State (Estat del Servei):** estat del servei en qualsevol moment de l'execució. Qualsevol informació que s'hagi de recuperar, modificar o supervisar en un servei DSS ha de formar part del seu estat.
- **Service Partners (Serveis Associats):** serveis especials que interaccionen amb d'altres i s'encarreguen de que funcionin correctament. Es declaren amb l'atribut "*Partner*".
- **Main Port (Port Principal):** És un port CCR on arriben els missatges d'altres serveis. Aquest port és un membre privat de la classe *Service* i s'identifica amb l'atribut *ServicePort*.
- **Service Handlers (Administradors de Servei):** registre de totes les operacions DSS definides sobre el port principal. Es defineixen amb l'atribut *ServiceHandlers*.

D'aquesta forma un robot no és una entitat en sí mateixa dins de Microsoft Robotics Studio, si no que és un conjunt de serveis, cadascun d'ells representant algun element del mateix, com pot ser un servei per a representar les seves rodes, un altre per a representar un sensor com pot ser una càmera, etc. Només es fan servir els serveis requerits. Això es veu clarament a Visual Programming Language, on cada instància de servei fa referència a una funció.

MRS permet construir funcions d'alt nivell fent servir components simples, proveint la possibilitat de reutilitzar els mòduls de codi, així com d'una millor fiabilitat i possibilitat de substitució. Per exemple, es pot integrar un servei de sensor de baix nivell a un servei de navegació.

Plataforma escalable i extensible

El model de programació de MRS es pot aplicar a una gran varietat de plataformes de hardware, donant la possibilitat als usuaris de transferir les seves habilitats entre plataformes. Les interfícies de programació es poden utilitzar per a desenvolupar aplicacions per a robots utilitzant processadors de 8,16 o 32 bits, tant d'un o més nuclis.

Les third parties poden estendre la funcionalitat de MRS proveint llibreries i serveis addicionals. Els comerciants de hardware o software poden fer els seus productes fàcilment compatibles amb MRS.

2.1.7. Llenguatges que es poden utilitzar en MRS

MRS permet que les aplicacions es desenvolupin en una gran varietat de llenguatges, incloent aquells que hi ha a Microsoft Visual Studio i Microsoft Visual Studio Express (C# i VB.NET), així com llenguatges com Microsoft Iron Python.

També permet llenguatges de third parties que suportin l'arquitectura basada en serveis del MRS.

En aquest projecte s'ha optat per utilitzar C#.

2.1.8. Versions

Actualment existeixen dues versions:

10. **Microsoft Robotics Studio 1.5:** és la versió que s'utilitzarà per a portar a terme aquest projecte ja que era la més avançada en els primers mesos de realització d'aquest. És una versió final, evolució de la versió 1.0 sortida al mercat el 2006.

- **Community Technical Preview July:** aquesta versió és una primera mostra de la nova evolució del programa. Es pot descarregar, però no és una versió final, és una versió pre-beta.
- **Microsoft Robotics Studio Developer Studio 2008 R2:** és la versió final d'aquesta nova evolució.

Microsoft Robotics Studio pot instal·lar-se tant en Windows XP com en Windows Vista.

2.1.9. Llicència

Existeixen dos tipus de llicència:

No Comercial

Llicència per a una utilització no comercial. Es defineix "no comercial" com a utilitzar el software per a ús personal i no per a fer negocis o generar beneficis. És una llicència indicada per a estudiants, professors, recerca acadèmica o gent que s'interessi com a hobby.

El software es pot descarregar gratuïtament. Les companyies o gent que pretén obtenir beneficis pot utilitzar aquest software per un període de temps limitat per a avaluar-ho.

La llicència no comercial només permet la utilització de les rutines de components en un nombre limitat de robots o PCs. El codi de mostra que inclou el software pot ser utilitzat, modificat i distribuït sempre dins els termes de la llicència.

Comercial

Aquesta llicència està indicada si es desitja crear o distribuir aplicacions comercials o basar operacions de negocis en Microsoft Robotics Studio. En aquests casos s'ha de comprar la llicència a la botiga online. Una llicència comercial costa 499\$ (per Espanya costa 535€, depenen del canvi de moneda actual).

La llicència comercial substitueix la no comercial i dóna dret a alguns beneficis, com una llicència per a distribuir els components de rutina amb el software creat. Cada llicència comercial permet distribuir 200 còpies d'aquests components. Si es disposava d'una llicència per Microsoft Robotics Studio 1.0, Microsoft considera la versió 1.5 com una actualització de la 1.0 respecte la versió 1.0 comercial adquirida prèviament. Microsoft permet que en cas de no haver distribuït les 200 rutines per la versió 1.0 es puguin distribuir les restants amb la versió 1.5, amb els mateixos termes de la llicència 1.0. No es poden sobrepassar les 200.

El codi de mostra que s'inclou es pot modificar i distribuir d'acord als termes de llicència.

2.2. Microsoft Visual c#

2.2.1 Introducció

Com que MRS treballa sobre .NET Framework es pot triar qualsevol dels llenguatges que dona suport a aquest. La major part d'exemples i tutorials disponibles en MRS són escrites en C# o Visual Bàsic .NET, per això per fer aquest projecte s'ha optat per el llenguatge C#.

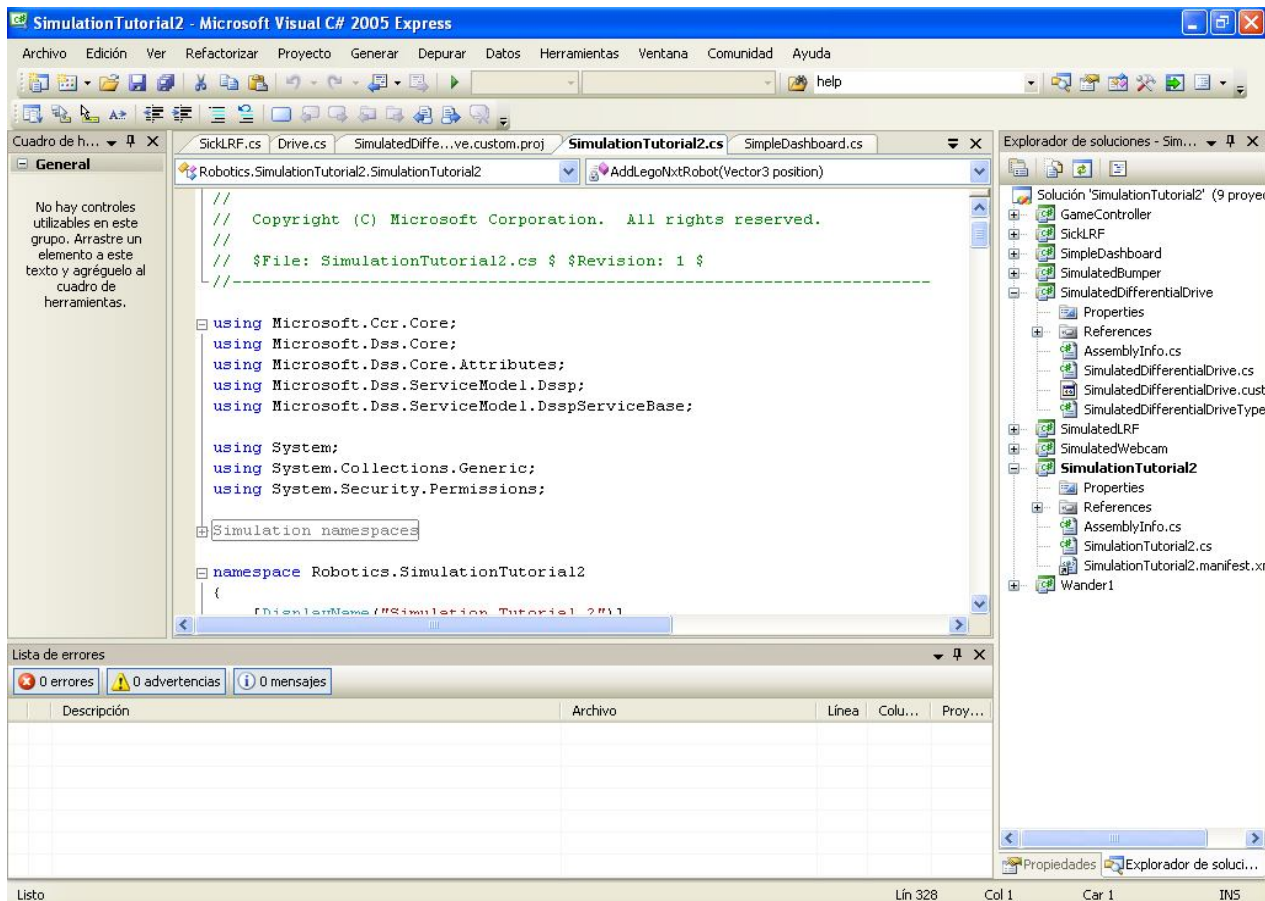


Figura 2.7: Interfície de programació Microsoft Visual C# 2005

Microsoft Visual C# és un llenguatge de programació dissenyat per crear una àmplia gamma d'aplicacions que s'executen en .NET. C# és simple, eficaç, amb seguretat de tipus i orientat a objectes. Amb les seves innovacions, C# permet desenvolupar aplicacions ràpidament i manté l'expressivitat i elegància dels llenguatges de tipus C.

Visual Studio C# és un editor de codi complet, plantilles de projecte, dissenyadors, assistents per a codi, un depurador eficaç i fàcil d'usar, a més d'altres eines (figura 2.7). La biblioteca de classes .NET Framework ofereix accés a una àmplia gamma de serveis de sistema operatiu i a altres classes útils i adequadament dissenyades que acceleren el cicle de desenvolupament de manera significativa.

2.2.2. Principals característiques

A continuació es recull de manera resumida les principals característiques de C#. No es detalla completament totes les característiques del llenguatge, sinó que la intenció és mostrar una visió general del llenguatge:

- Té totes les característiques pròpies de qualsevol llenguatge orientat a objectes: encapsulació, herència i polimorfisme.

- Ofereix un model de programació orientada a objectes homogeni, en el qual tot el codi s'escriu dins de classes i tots els tipus de dades, fins i tot els bàsics, són classes que hereten de System.Object (pel qual els mètodes definits en aquesta són comuns a tots els tipus del llenguatge).
- Permet definir estructures, que són classes un tant especials; els seus objectes s'emmagatzemen en pila, pel que es treballa amb ells directament i no referències en monticle, el que permet accedir-los més ràpid. Tanmateix, aquesta major eficiència en els seus accessos té també els seus inconvenients, fonamentalment que el temps necessari per passar-les com a paràmetres a mètodes és major (cal copiar el seu valor complet i no només una referència) i no admeten herència (encara que sí implementació d'interfícies).
- Es un llenguatge fortament de tipus, el que significa que controla que totes les conversions entre tipus es realitzin de forma compatible, el que assegura que mai s'accedeixi fora de l'espai de memòria ocupat per un objecte. Així s'eviten freqüents errors de programació i s'aconsegueix que els programes no puguin posar en perill la integritat d'altres aplicacions.
- Té a la seva disposició un recol·lector d'escombraries que alliberen el programador de la tasca d'haver d'eliminar les referències a objectes que deixin de ser útils, encarregant-se'n aquest i evitant així que s'esgoti la memòria perquè al programador se li oblidí alliberar objectes inútils o que es produeixin errors perquè el programador alliberi àrees de memòria ja alliberades i reassignades.
- Inclou suport natiu per a esdeveniments i delegats. Els delegats són similars als punters a funcions d'altres llenguatges com a C++ encara que més propers a la orientació a objectes, i els esdeveniments són mecanismes mitjançant els quals els objectes poden notificar de l'ocurrència d'esdeveniments. Els esdeveniments solen utilitzar en combinació amb els delegats per al disseny d'interfícies gràfiques d'usuari, amb el que es proporciona al programador un mecanisme còmode per escriure codis de resposta als diferents esdeveniments que puguin sorgir al llarg de l'execució de l'aplicació (pulsació d'un botó, modificació d'un text, etc.).
- Incorpora propietats, que és un mecanisme que permet l'accés controlat a membres d'una classe tal i com si de camps públics es tractés. Gràcies a elles s'evita la pèrdua de llegibilitat que en altres llenguatges causa la utilització de mètodes Set() i Get() però es mantenen tots els avantatges d'un accés controlat per aquests.
- Permet la definició del significat dels operadors bàsics del llenguatge (+, -, *, &, ==, etc.) per als nostres propis tipus de dades, el que facilita enormement tant la llegibilitat de les aplicacions com l'esforç necessari per escriure-les. És pot fins i tot definir el significat de l'operador [] en qualsevol classe, el que permet accedir als seus objectes tal i com si fossin taules. A la definició d'aquest últim operador se li denomina indicador, i és especialment útil a l'hora d'escriure o treballar amb col·leccions d'objectes.
- Admet uns elements anomenats atributs que no són membres de les classes sinó informació sobre aquestes que podem incloure en la seva declaració. Per exemple, indiquen si un membre d'una classe ha d'aparèixer a la finestra de propietats de Visual Studio.NET, quins són els valors admesos per a cada membre en aquesta...

2.3 Robot E-puck

2.3.1. Descripció general

El robot e-puck (veure figura 2.8) va ésser concebut com a un robot de baix cost orientat a docència i investigació en robòtica mòbil. Les dimensions del robot són de 7 cm de diàmetre i 4.2 cm d'alçada. L'estructura del robot és d'una sola peça on es poden encabir totes les parts que el formen.



Figura 2.8: Robot e-puck

A la part superior de l'estructura s'hi col·loca el circuit imprès; a l'interior, dos motors; i, per últim, a sota dels motors s'hi troba la bateria que es pot extreure amb facilitat. El conjunt dels sensor de proximitat i llum van col·locats al voltant del circuit imprès i la càmera se situa en un petit suport que hi ha sota la placa a la part frontal. Podem veure aquesta estructura a la figura 2.9.

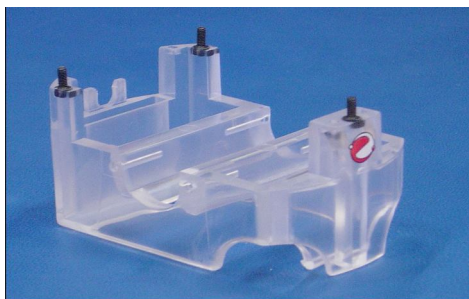


Figura 2.9: Estructura del robot

El sistema d'eixos que utilitzem és el següent: l'eix X és el que segueix el moviment del robot e-puck, l'eix Y perpendicular amb l'eix X i segueix la direcció de l'eix de les rodes i, per últim, l'eix Z va ortogonal als eixos X i Y anant en el sentit del terra cap al sostre. Veiem la figura 2.10.

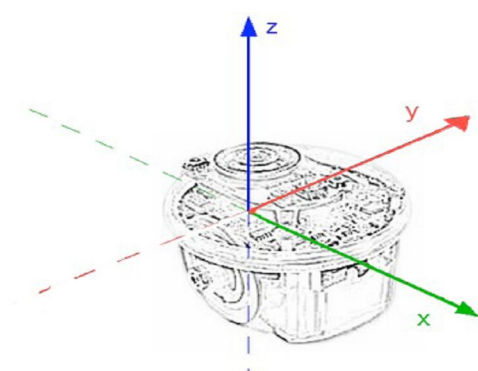


Figura 2.10: Eixos del robot e-puck

L'encarregat de controlar tots els dispositius és un dsPIC, concretament el model 30F6014A. Les característiques principals són: la mida de la paraula és de 16 bits, utilitza una memòria de tipus Flash. La memòria del programa és de 144KB i disposa de 8 MB de RAM com es veu a la figura 2.11.

Feature	Technical information
Size, weight	70 mm diameter, 55 mm height, 150 g
Battery autonomy	5Wh LiION rechargeable and removable battery providing about 3 hours autonomy.
Processor	dsPIC 30F6014A @ 60 MHz (~15 MIPS) 16 bits microcontroller with DSP core
Memory	RAM: 8 KB; FLASH: 144 KB
Motors	2 stepper motors with a 50:1 reduction gear, resolution: 0.13 mm
Speed	Max: 15 cm/s
Mechanical structure	Transparent plastic body supporting PCBs, battery and motors
IR sensors	8 infra-red sensors measuring ambient light and proximity of objects up to 6 cm
Camera	VGA color camera with resolution of 640x480 (typical use: 52x39 or 640x1)
Microphones	3 omni-directional microphones for sound localization
Accelerometer	3D accelerometer along the X, Y and Z axis
LEDs	8 red LEDs in the ring, green LEDs in the body, 1 strong red LED in front
Speaker	On-board speaker capable of WAV or tone sounds playback
Switch	16 position rotating switch on the top of the robot
PC Connection	Standard serial port up to 115kbps
Wireless	Bluetooth for robot-computer and robot-robot wireless communications
Remote control	Infra-red receiver for standard remote control commands
Expansion bus	Large expansion bus designed to add new capabilities
Programming	C programming with the free GNU GCC Graphical IDE (integrated development environment) provided in Webots
Simulation	Webots facilitates the use of the e-puck: powerful simulation, remote control and C programming system.

Figura 2.11: Especificacions del robot e-puck

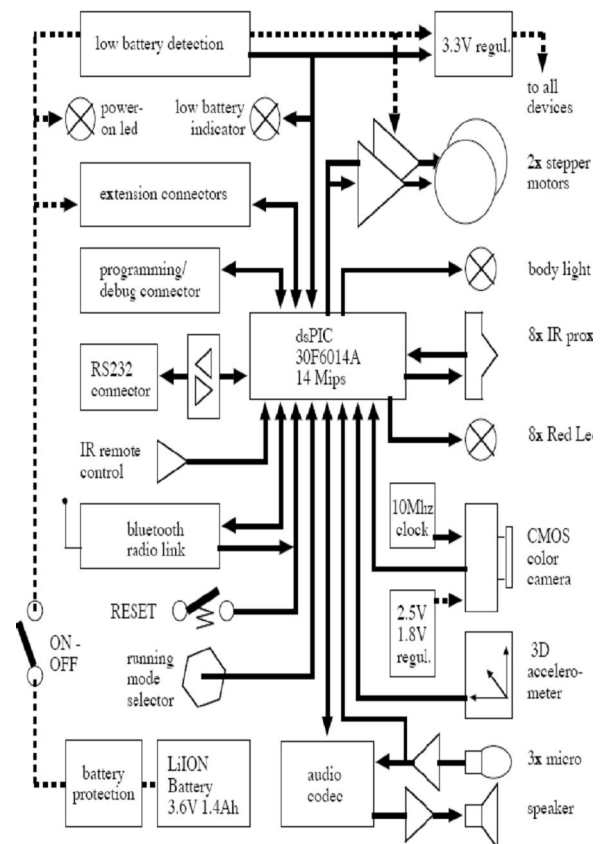


Figura 2.12: Esquema elèctric del robot e-puck

Per tal d'obtenir una visió general de tots els elements de què disposa l'e-puck, podem consultar-ne l'esquema elèctric (veure figura 2.12), on es pot apreciar tot el conjunt de sensors i actuadors que es descriuen a les seccions següents.

En el següent esquema (figura 2.13) es pot apreciar on estan situades totes les parts.



Figura 2.13: Situació dels elements del e-puck

2.3.2. Sensors

Els sensors són els dispositius que, a partir de l'energia del medi o d'una energia pròpia, donen un senyal (normalment elèctric) de sortida que sol ser una determinada funció de la variable que volem mesurar. Utilitzant els sensors podem mesurar fenòmens físics com poden ser la velocitat, l'acceleració, la distància... El robot e-puck disposa d'un ventall molt ampli de sensors per tal de poder detectar qualsevol canvi en el seu entorn més immediat i en ell mateix.

A continuació s'expliquen tots els sensors disponibles en l'e-puck.

● Sensors Infrarojos

Aquests sensors detecten la proximitat dels objectes que envolten l'e-puck. El funcionament d'aquests sensors consisteix en enviar primer un feix de llum infraroja i mesurar després la quantitat de llum que retorna. Si algun objecte es troba en el camí d'aquest feix, part de la llum s'hi reflectirà i tornarà al sensor. Segons la proximitat de l'objecte i la seva reflectivitat arribarà més o menys llum.

Amb la intenció de tenir el màxim control de tota la zona al voltant del robot els sensors estan col·locats d'una manera estratègica. A la taula 2.1 podem veure els angles que corresponen a cada sensor respecte l'eix X del sistema de coordenades del robot e-puck que hem definit anteriorment, i la seva posició en la Figura 2.14:

Sensors	Orientació respecte la direcció d'avanç (°)
IR0	-17,2344
IR1	- 45,8823
IR2	-90
IR3	-151,489
IR4	156,9448
IR5	90
IR6	45,79
IR7	17,1431

Taula 2. 1: Sensors IR i la seva orientació

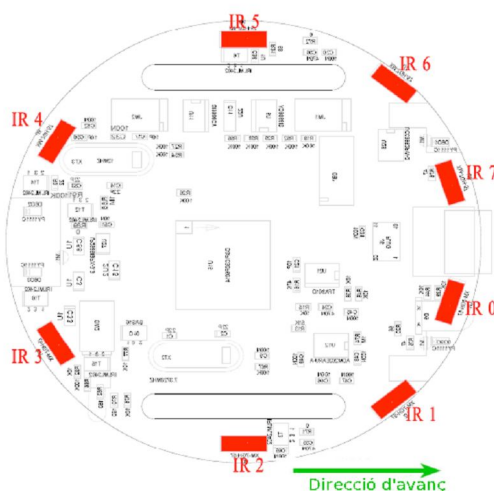


Figura 2.14: Situació dels sensor IR del robot e-puck

Els valors que s'obtenen d'aquest sensor, per conèixer la distància de l'objecte que detecten, es calculen a través d'una transformació. El sensor no té un comportament lineal i per aquesta raó podem dividir la funció en diferents rectes. La taula 2.2 mostra els valors de cada tram.

Distància(cm)	Valor Sensor	Sensor – Soroll	Sensor + Soroll	Soroll(%)
0	4095	4074,53	4115,48	0,5
0,5	3474	3345,46	3602,54	3,7
1	2211	2054,02	2367,98	7,1
2	676	605,02	746,98	10,6
3	306	267,75	344,25	12,5
4	164	130,22	197,78	20,6
5	90	65,79	114,21	26,9
6	56	31,47	80,53	43,8
7	34	10,06	57,94	70,4

Taula 2. 2: Equivalència entre la distància i el valor que obté el sensor

La gràfica figura 2.15 que obtenim és la següent:

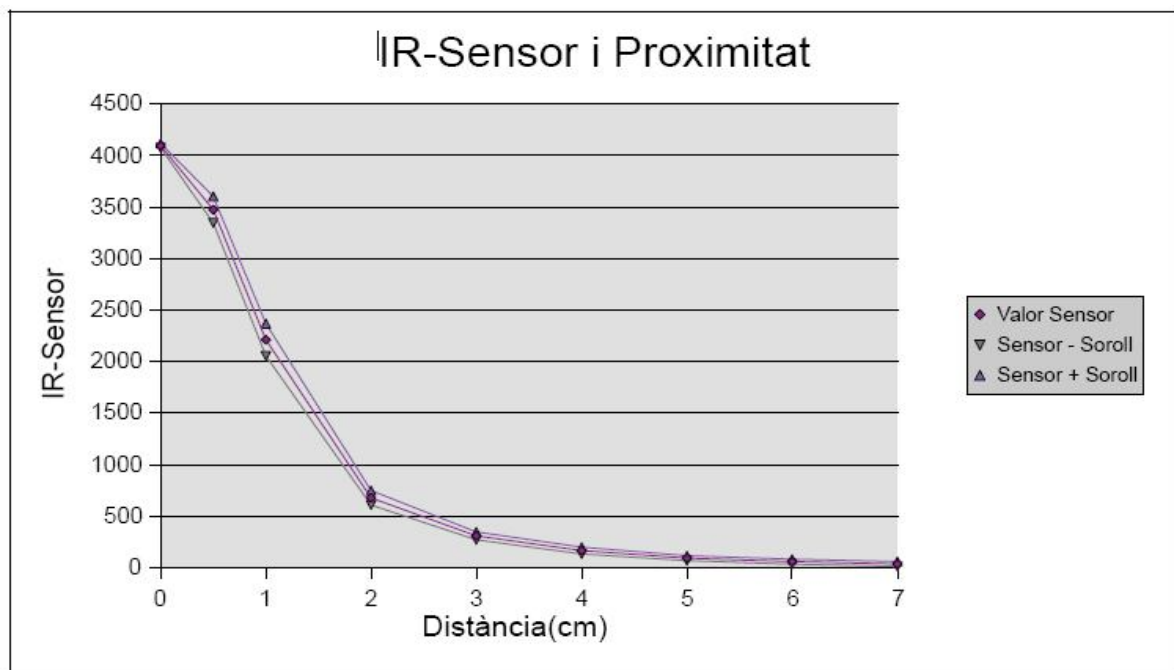


Figura 2.15: Gràfica de relació dels valors obtinguts per un sensor IR i la Distància que es troba l'objecte

- **Encoders**

El robot disposa de dos encoders que permeten conèixer la posició de cada roda del robot. En realitat no es disposa d'aquests sensors sinó que s'emulen en el firmware del propi robot. Això és possible perquè el robot utilitza dos motors pas a pas connectats a un reductor cadascun; aquests motors transformen un impuls elèctric en un determinat angle i això fa possible que es pugui emular els encoders.

- **Acceleròmetre**

El robot e-puck disposa d'un acceleròmetre que mesura les acceleracions lineals als tres eixos XYZ. A la figura 2.16 es pot observar a quina posició es troba l'acceleròmetre.

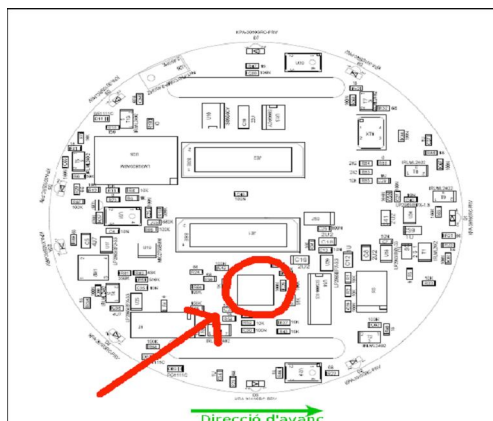


Figura 2.16: Posició de l'acceleròmetre del robot e-puck

- **Càmera**

Es disposa d'una càmera CMOS en color, amb una resolució d'imatge de 640x480. La càmera no es pot fer servir en el seva màxima potència, ja que el microprocessador no és prou potent ni tampoc tenim prou capacitat de memòria per guardar la imatge completa. Per aquesta raó, es recomana utilitzar una resolució de 40x40 amb 4 frames per segon. La càmera està situada a la part frontal del robot tal i com podem observa a la figura 2.17.



Figura 2.17: Situació de la càmera del robot e-puck

- **Micròfons**

El robot té tres micròfons situats d'una manera estratègica. Aquesta disposició permet triangular els sons i conèixer aproximadament on es troba l'origen del so respecte al robot. La màxima freqüència d'adquisició per cada micròfon és de 33KHz. A continuació, podem veure a la figura 2.18 quina disposició tenen els tres micròfons.

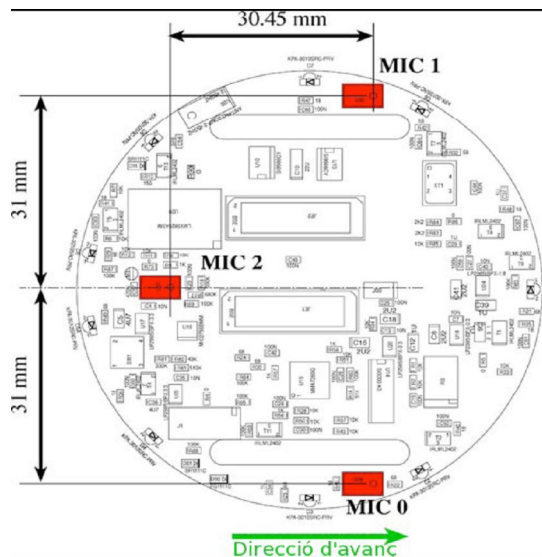


Figura 2.18: Situació dels micròfons en el robot e-puck

2.3.3. Actuadors

Els actuadors són dispositius que, a partir d'una energia normalment elèctrica, són capaços de modificar una determinada magnitud física. Utilitzant diferents tipus d'actuadors el robot e-puck pot interaccionar amb l'entorn. Disposa de tres tipus d'actuadors: lumínics, sonors i de moviment.

- **Leds**

El robot e-puck disposa de vuit leds col·locats sobre els sensors de proximitat (a l'anell superior que envolta el circuit imprès). Aquest leds són tots de color vermell. També disposa d'un led intern de color groc que il·lumina el cos del robot i, finalment, té també un led vermell més potent al costat de la càmera. A la figura 2.19 podem observar el robot amb tots els leds actius i a la figura 2.20 tenim marcat l'anell de leds amb uns quants leds marcats i el led frontal.



Figura 2.19: Robot e-puck amb tots els leds encesos.



Figura 2.20: Situació dels leds del robot e-puck

- **Altaveu**

Es disposa d'un únic altaveu col·locat a la part superior del robot, des del qual es poden emetre sons. Es tracta de sons pensats per anar de l'altaveu als micròfons d'un altre robot per tal que aquest els pugui interpretar i poder arribar a comunicar-se. Vegeu la figura 2.21 per comprovar on està situat l'altaveu.



Figura 2.21: Situació de l'altaveu del robot e-puck

- **Motor**

El robot disposa de dos motors pas a pas de 20 passos per revolució connectats cadascun d'ells amb un reductor. La roda està connectada a l'eix que surt del reductor. El reductor té una relació de 50:1, on sigui, per cada 50 revolucions del motor pas a pas la roda realitza una volta. Per tant, cada 1000 passos del motor la roda haurà fet una volta. (Figura 2.22)

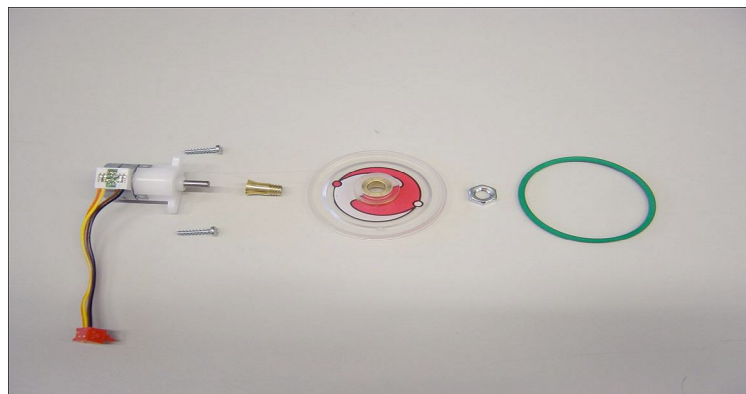


Figura 2.22: Motor i rodes del robot e-puck desmuntats

2.3.4. Connectivitat i comunicació

Per connectar amb el robot tenim dues opcions possibles: una opció és utilitzant la connexió bluetooth mitjançant el receptor bluetooth i l'altra opció és utilitzant un connector port sèrie RS232.

El bluetooth utilitza el xip LMX9820A. Aquest xip permet connectar-nos utilitzant l'uart amb un canal rfcmm de bluetooth. A través d'aquest canal ens comunicarem amb el robot com si es tractés d'un port sèrie amb l'excepció que els missatges de connexió i desconnexió són diferents. Per connectar-nos al robot, cal el codi PIN i s'ha d'introduir a cada connexió, aquest codi és el numero del robot.

La comunicació amb el robot e-puck es realitza mitjançant l'enviament i recepció de missatges. Aquests missatges estan compostos de caràcters, els missatges que s'envien al robot són del següent estil:

Tipus	,	(Paràmetres)	,	Final de missatge
-------	---	--------------	---	-------------------

On **tipus** és un caràcter que serà una lletra majúscula.

Paràmetres és el conjunt de paràmetres de tipus enter cada un escrit en una cadena de caràcters separats entre ells per el caràcter ',' Aquest camp és opcional.

Final de missatge és el caràcter 'r'.

Els missatges de resposta que respon el robot són del següent estil:

Tipus	,	(Paràmetres)	,	Final de missatge
-------	---	--------------	---	-------------------

On **tipus** és un caràcter que serà la mateixa lletra que la de petició però en minúscula .

Paràmetres és el conjunt de paràmetres de tipus enter cada un escrit en una cadena de caràcters separats entre ells per el caràcter ',' o un text depenent del tipus de resposta.

Aquest camp pot ser inexistent en els casos en que s'actualitzi un actuator.

Final de missatge és el caràcter 'r'.

A continuació s'expliquen totes les accions que es poden demanar al robot i els missatges necessaris per realitzar-les que s'han d'enviar al robot i les seves respostes segons el *advanced sercom protocol*.

Acceleròmetre: S'ha d'enviar "Av" i el robot respondrà "a,x,y,z\r\n" on x, y i z són enters que representen l'acceleració dels 3 eixos respectivament.

Led del cos: S'ha d'enviar "B,i\r\n" on "i" és l'estat que es vol posar al led, 0=apagat, 1=encès o 2=estat invers a l'actual. El robot respondrà "b\r\n" en acabar d'actualitzar el led.

Posició del selector: S'ha d'enviar "Cv" i el robot respondrà "c,x,\r\n" on "x" és un enter que representen la posició del selector del robot e-puck.

Posar la velocitat dels motors: S'ha d'enviar "D,x,y\r\n" on "x" és la velocitat del motor esquerre i y és la velocitat del motor dret. El robot respondrà "d\r\n" en acabar d'actualitzar els motors.

Llegir la velocitat dels motors: S'ha d'enviar "Ev". El robot respondrà "e,x,y\r\n" on "x" és la velocitat del motor esquerre i y és la velocitat del motor dret.

Led de frontal: S'ha d'enviar "F,i\r" on "i" és l'estat que es vol posar al led, 0=apagat, 1=encès o 2=estat invers a l'actual. El robot respondrà "f\r" en acabar d'actualitzar el led.

Receptor IR: S'ha d'enviar "G\r". El robot respondrà "g IR check : 0xX, address : 0xY, data : 0xZ\r" on X és el check, Y és l'adreça i Z és la dada.

Ajuda del robot: S'ha d'enviar "H\r". El robot respondrà un text que informa sobre les opcions que s'expliquen aquí.

Obtenir els paràmetres de la càmera: S'ha d'enviar "I\r". El robot respondrà "i,X,Y,Z,T,U\r" on X és el mode de la càmera, Y és l'amplada, Z és l'altura, T és el zoom i U és la mida.

Posar els paràmetres de la càmera: S'ha d'enviar "J,X,Y,Z,T,U,\r" on X és el mode, Y és l'amplada, Z és l'altura, T és el zoom i U és la mida. El robot respondrà "j,\r".

Calibrar els sensors de proximitat: S'ha d'enviar "K\r". I el robot respondrà "k, Starting calibration - Remove any object in sensors range\r" abans de començar a calibrar i en acabar enviarà "k, Calibration finished\r".

Anell de leds: S'ha d'enviar "L,I,J\r" on "I" és el numero del led que es vol modificar i "J" és l'estat que es vol posar al led, 0=apagat, 1=encès o 2=estat invers a l'actual . El robot respondrà "l\r" en acabar d'actualitzar el led.

Llegir els sensors de proximitat: S'ha d'enviar "N\r". El robot respondrà "n,S0,S1,S2,S3,S4,S5,S6,S7\r" on Si és el valor del sensor de proximitat i.

Llegir els sensors de lluminositat: S'ha d'enviar "O\r". El robot respondrà "o,S0,S1,S2,S3,S4,S5,S6,S7\r" on Si és el valor del sensor de lluminositat i.

Posar la posició dels motors: S'ha d'enviar "P,X,Y\r" on X és la posició del motor esquerre i Y és la posició del motor dret.. El robot respondrà "p,\r" en acabar.

Llegir la posició dels motors: S'ha d'enviar "Q\r". El robot respondrà "q,X,Y\r" on X és la posició del motor esquerre i Y es la posició del motor dret.

Resetejar l'e-puck: S'ha d'enviar "R\r". El robot respondrà "r,\r" en acabar.

Parar l'e-puck: S'ha d'enviar "S\r". El robot respondrà "s,\r" en acabar de parar els motors i apagar els leds.

Reproduir un so: S'ha d'enviar "T,X,\r" reproduïx el so X si X es un nombre entre 1 i 5, altrament apaga l'altaveu. El robot respondrà "t,\r" en acabar.

Llegir l'amplitud del micròfon: S'ha d'enviar "U\r". El robot respondrà "u,S0,S1,S2\r" on Si és l'amplitud del micròfon i.

Obtenir la versió del controlador: S'ha d'enviar "V\r". El robot respondrà "v,version\r" on *version* és un text on diu la versió del controlador.

3.SIMULACIÓ DEL ROBOT E-PUCK

3.1. Introducció

La simulació del robot e-puck és una de les parts més importants del projecte ja que tenir una bona simulació permet poder treballar i crear programes que puguin controlar un robot real, sense necessitat de tenir-lo durant el procés de disseny i programació.

Després d'investigar com es simulen els robots mòbils estudiant les simulacions existents dins el paquet de MRS es va concloure que totes seguien el mateix sistema. Primer es crea l'aspecte físic de l'entorn de simulació i a continuació s'hi afegeixen les entitats.

3.2 Implementació

Per implementar la simulació s'han realitzat els següents passos:

3.2.1 Crear el projecte

El primer que es va fer va ser crear un nou projecte que s'ha anomenat *simEpuck* utilitzant la plantilla de *SimpleDssService*. Això crearà un projecte amb dos fitxers de classe *simEpuck.cs* on s'escriurà el codi que s'executarà i *simEpuckTypes.cs* que no s'ha modificat.

3.2.2 Afegir components

A continuació s'ha afegit les referències necessàries per poder accedir a elements necessaris per fer una simulació com es pot veure a la taula 3.1:

Nombre del component	Descripció
PhysicsEngine	Proporciona accés al motor físic del software AGEIA subjacent.
RoboticsCommon	Proporciona accés al espai de noms PhysicalModel, que s'utilitza per definir les característiques físiques de los robots.
SimulationCommon	Proporciona accés a les definicions de tipus que s'utilitza al treballar amb la simulació i amb els motors físics.
SimulationEngine	Proporciona accés al motor de la simulació.
SimulationEngine.proxy	Representa un proxy per el motor de la simulació, que s'utilitza al carregar el motor de la simulació com associat.
Microsoft.Xna.Framework	S'utilitza per representar entitats.

Taula 3. 1: Components necessaris per la simulació

Després d'afegir les referències, era necessari afegir les següents declaracions de l'espai de noms al fitxer d'implementació de classe *SimEpuck.cs*:

```
using Microsoft.Robotics.Simulation.Engine;  
using engineproxy = Microsoft.Robotics.Simulation.Engine.Proxy;  
using Microsoft.Robotics.Simulation.Physics;  
using Microsoft.Robotics.PhysicalModel;  
using xna = Microsoft.Xna.Framework;
```

El següent pas és modificar el mètode *protected override void Start()*

```
protected override void Start()
{
    // Listen on the main port for requests and call the appropriate handler.
    ActivateDsspOperationHandlers();

    // Publish the service to the local Node Directory
    DirectoryInsert();

    // Cache references to simulation/rendering and physics
    _physicsEngine = PhysicsEngine.GlobalInstance;
    _simEnginePort = SimulationEngine.GlobalInstancePort;

    //Set up initial view
    SetupCamera();

    // Add objects (entities) in our simulated world
    PopulateWorld();
}
```

A continuació s'han creat els mètodes *SetupCamera()* que determina els paràmetres de la càmera dintre la simulació, i el mètode *PopulateWorld()* que crida els mètodes per afegir el cel *AddSky()*, el terra *AddGround()*, les parets per simular l'entorn de pràctiques amb el robot real del laboratori de robòtica *AddWalls()*, i per últim el mètode per afegir el robot e-puck *AddEpuck()*.

Els mètodes *SetupCamera()*, *AddSky()* i *AddGround()* s'han extret del projecte *SimulationTutorial2*.

El mètode *AddWalls()* crida a quatre mètodes que cadascun d'ells afegeix a la simulació una entitat de tipus caixa de forma que entre les quatre formen un habitacle de les mateixes dimensions que l'entorn de pràctiques real com es veu a la figura 3.1.

```
private void AddWall80()
{
    BoxShapeProperties cBoxShape = null;//
    SingleShapeEntity cBoxEntity = null;
    cBoxShape = new BoxShapeProperties(10f, new Pose(), new Vector3(1, 0.1f, 0.1f)); //massa,pose,dimension(x,y,z)
    cBoxShape.Material = new MaterialProperties("gbox", 1, 1, 1);
    cBoxShape.DiffuseColor = new Vector4(0.1f, 0.5f, 0.4f, 1.0f);
    cBoxEntity = new SingleShapeEntity(new BoxShape(cBoxShape), new Vector3(0, 0.1f, -0.35f)); //Posicio
    cBoxEntity.State.Name = "greybox";
    SimulationEngine.GlobalInstancePort.Insert(cBoxEntity);
}
```

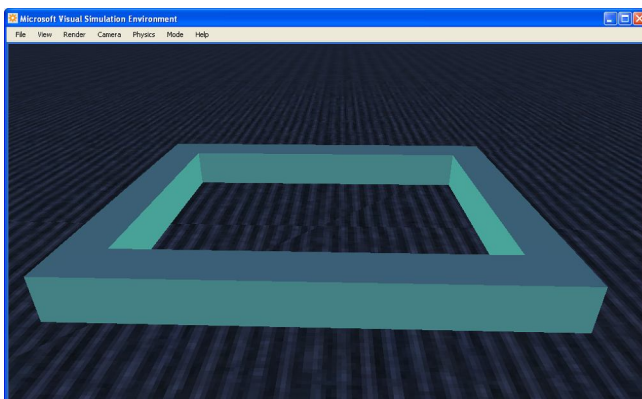


Figura 3.1: A l'esquerre simulació d'entorn de treball a la dreta entorn de treball real

3.2.3 Creació d'una classe identitat per al robot e-puck:

L'últim pas però el més important és crear una nova entitat de robot per representar l'e-puck, per fer això és necessari afegir una nova classe derivada de *DifferentialDriveEntity* al projecte que s'ha anomenat *Epuck.cs*.

Al derivar-la de la classe *DifferentialDriveEntity*, és possible tornar a utilitzar el codi que defineix el comportament del e-puck quan es mogui en una simulació. El codi per crear aquest tipus d'entitat del e-puck és el següent:

```
using Microsoft.Dss.Core.Attributes;
using Microsoft.Ccr.Core;
using Microsoft.Robotics.Simulation.Physics;
using Microsoft.Robotics.Simulation.Engine;
using Microsoft.Robotics.PhysicalModel;

namespace Robotics.SimEpuck
{
    [DataContract]
    public class Epuck : DifferentialDriveEntity
    {
        /// <summary>
        /// Default constructor
        /// </summary>
        public Epuck() {}

        /// <summary>
        /// Initialization constructor
        /// </summary>
        /// <param name="initialPos"> es la posicio inicial del robot </param>
        public Epuck(Vector3 initialPos)
        {
            MASS = 0.150f; //Pes en kg

            CHASSIS_DIMENSIONS = new Vector3(0.07f, 0.055f, 0.07f); //Dimensions x,y,z en metres
            CHASSIS_CLEARANCE = 0.002f; //distància del chasis a terra

            FRONT_WHEEL_MASS = 0.01f; //Pes de les rodes
            FRONT_WHEEL_RADIUS = 0.0205f; //Radi de les rodes

            CASTER_WHEEL_RADIUS = 0.0002f;
            FRONT_WHEEL_WIDTH = 0.028f; //no s'utilitza
            CASTER_WHEEL_WIDTH = 0.002f; //no s'utilitza
            FRONT_AXLE_DEPTH_OFFSET = -0.001f; // distance of the axle from the center of robot

            base.State.Name = "Epuck";
            base.State.MassDensity.Mass = MASS;
            base.State.Pose.Position = initialPos;

            // reference point for all shapes is the projection of
            // the center of mass onto the ground plane
            // (basically the spot under the center of mass, at Y = 0, or ground level)

            // chassis position
            BoxShapeProperties motorBaseDesc = new BoxShapeProperties("chassis", MASS,
                new Pose(new Vector3(
                    0, // Chassis center is also the robot center, so use zero for the X axis offset
                    CHASSIS_CLEARANCE, // Chassis center is also the robot center, so use zero for the Y axis
                    0.001f)), // no offset in the z/depth axis, since again, its center is the robot center
                CHASSIS_DIMENSIONS);
        }
    }
}
```

```

ChassisShape = null;

motorBaseDesc.Material = new MaterialProperties("high friction", 0.05f, 0.05f, 0.05f);
motorBaseDesc.Name = "Chassis";
ChassisShape = new BoxShape(motorBaseDesc);

// rear wheel is also called the caster
CASTER_WHEEL_POSITION = new Vector3(0, // center of chassis
    -CHASSIS_DIMENSIONS.Y/2, // distance from ground
    CHASSIS_DIMENSIONS.Z / 2 - CASTER_WHEEL_RADIUS); // all the way at the back of the
robot

// NOTE: right/left is from the perspective of the robot, looking forward

FRONT_WHEEL_MASS = 0.10f;

RIGHT_FRONT_WHEEL_POSITION = new Vector3(
    CHASSIS_DIMENSIONS.X / 2 - 0.053f, // left of center
    FRONT_WHEEL_RADIUS - CHASSIS_DIMENSIONS.Y / 2, // distance from ground of axle
    FRONT_AXLE_DEPTH_OFFSET); // distance from center, on the z-axis

LEFT_FRONT_WHEEL_POSITION = new Vector3(
    -CHASSIS_DIMENSIONS.X / 2 + 0.053f, // right of center
    FRONT_WHEEL_RADIUS - CHASSIS_DIMENSIONS.Y / 2, // distance from ground of axle
    FRONT_AXLE_DEPTH_OFFSET); // distance from center, on the z-axis

MotorTorqueScaling = 10;
State.Assets.Mesh = "miniepuck2.bos"; //Introduim la imatge del robot
}

public override void Initialize(Microsoft.Xna.Framework.Graphics.GraphicsDevice device,
PhysicsEngine physicsEngine)
{
    base.Initialize(device, physicsEngine);
}

public void Subscribe(Port<EntityContactNotification> notificationTarget)
{
    PhysicsEntity.SubscribeForContacts(notificationTarget);
}
}
}

```

S'utilitza un constructor per la classe *Epuck* per tal de configurar els valors de les variables que es defineixen a la classe *DifferentialDriveEntity*. Per exemple, la massa serà de 150g. També es defineix el chasis del robot segons l'amplada, longitud i alçada.

La posició del robot es defineix mitjançant una sèrie de coordenades que es transmeten quan es crea l'entitat. Aquestes coordenades representen els punts de l'eix x, y i z. El motor de simulació de MRS utilitza un sistema de coordenades de ma dreta el que afecta a la direcció cap on apunta l'eix Z.

El constructor també defineix la posició del chasis i les rodes de la entitat. La classe *DifferentialDriveSystem* assumeix que el robot té dos rodes principals i una roda posterior més petita que s'usa per mantenir l'equilibri. El funcionament s'assigna als motors de l'esquerre i de la dreta, que controlen les rodes principals. La diferència entre els nivells de funcionament assignats a cada roda determina si es mou endavant, endarrere, a l'esquerra o a la dreta.

L'atractiu de la simulació resideix que, teòricament, no importa si el robot és de debò o no: el codi que s'usa perquè funcioni el robot i per a enviar dades als sensors és el mateix. Una part del codi que s'usa en el projecte de simulació es pot tornar a usar al treballar amb el robot de debò. Que funcioni teòricament es deu al fet que la simulació no pot simular un entorn real per complet. La simulació no pot tenir en compte el soroll: les coses inesperades com els obstacles que es troben en el lloc equivocat. El que sí pot fer una simulació és proporcionar-li una oportunitat bastant realista d'experimentar amb el disseny d'un nou robot o simular una interacció entre diversos d'ells. Això té una enorme utilitat en entorns acadèmics en els quals els recursos són limitats i el nombre d'estudiants és elevat.

3.2.4 Creació d'una malla

Cada entitat es pot associar a una malla, que és el que fa que una entitat sembli realista. En el nostre cas la malla es el que fa que la forma cubica que té l'entitat del robot s'assembli al robot real d'una forma cilíndrica. Per assignar la malla al robot s'ha utilitzat la següent instrucció `State.Assets.Mesh = "miniepuck2.bos"`; que assigna el fitxer "miniepuck2.bos" a la imatge del robot. En un sentit estricte, no es necessari associar una malla a una entitat, però en el cas de les entitats complexes, com és el cas dels robots, és preferible usar un objecte de malla com es pot apreciar a les figures 3.2 i 3.3.

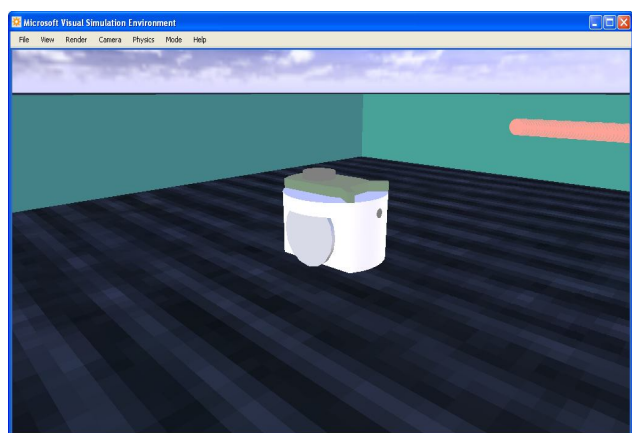
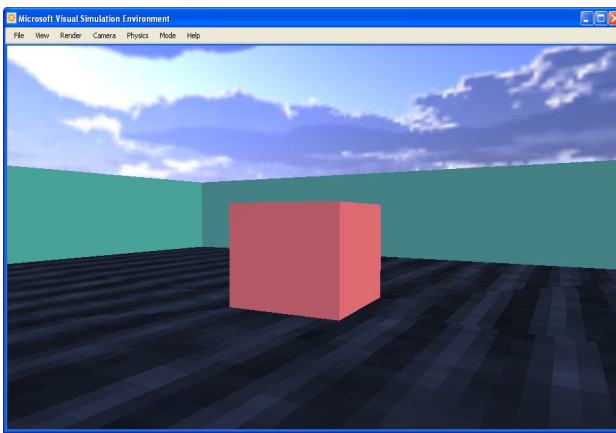


Figura 3.2: Simulació del robot sense malla **Figura 3.3:** Simulació del robot amb malla

Per a crear una malla, es poden usar gairebé totes les eines d'edició de gràfics en 3D. Per a aquest projecte vaig tenir la sort de comptar amb un col·lega, David Ribas del laboratori de robòtica submarina, que va crear la malla del e-puck en un format .obj. Un cop es té la malla en aquest format estàndard en programes d'edició 3D s'ha de guardar en el directori `\store\media` que està associat a la instal·lació local de MRS. Per poder utilitzar aquesta malla primer s'ha de convertir a un fitxer binari optimitzat amb una extensió .bos. Per fer això MRS ofereix una eina de conversió que s'hi accedeix mitjançant l'element de menú de símbol de sistema que s'inclou en instal·lació de MRS, un cop oberta la finestra de comandes s'ha introduït la següent comanda:

```
Obj2bos.exe /i:"store\media\miniepuck2.obj"
```

La eina de conversió crea un fitxer anomenat `miniepuck2.bos`, que es guarda al mateix directori al qual s'hi accedeix directament per carregar la malla.

Un cop acabada la classe que defineix el robot e-puck s'ha de crear el servei per poder-hi accedir, la següent part del codi és la que fa aquesta acció.

```
drive.Contract.CreateService(ConstructorPort,
    Microsoft.Robotics.Simulation.Partners.CreateEntityPartner(
        "http://localhost/" + EpuckBaseEntity.State.Name)
    );
```

3.2.5 Webcam

Després d'això ja tindriem el robot amb els motors, a continuació s'hi afegeix una càmera fixa al robot per tal de representar la webcam del robot real. Per fer això creem una entitat de tipus càmera amb les dimensions i propietats de la real i finalment es crea el servei per accedir-hi.

```
private CameraEntity CreateEpuckCamera()
{
    // low resolution, wide Field of View
    CameraEntity cam = new CameraEntity(320, 240);
    cam.State.Name = "Epuckcam";
    // just on top of the bot
    cam.State.Pose.Position = new Vector3(0.0f, 0.025f, 0.0f);
    // camera renders in an offline buffer at each frame
    // required for service
    cam.IsRealTimeCamera = true;

    // Start simulated webcam service
    simwebcam.Contract.CreateService(
        ConstructorPort,
        Microsoft.Robotics.Simulation.Partners.CreateEntityPartner(
            "http://localhost/" + cam.State.Name)
    );

    return cam;
}
```

El codi que l'insereix en el robot és el següent:

```
// create Camera Entity ans start SimulatedWebcam service
CameraEntity camera = CreateEpuckCamera();
// insert as child of motor base
epuckBaseEntity.InsertEntity(camera);
```

3.2.6 Sensors IR

L'últim que s'ha decidit que contingui el robot dintre la simulació per tal de poder realitzar un bon nombre de simulacions és el conjunt de sensors d'infraroig per detectar distàncies. Al investigar els sensors sobre els que treballa MRS es va veure que no tenien previst aquest tipus de sensors, finalment es va decidir utilitzar dos *LaserRangeFinderEntity*, que és un sensor làser de 180° que retorna una taula de distàncies de les quals només utilitzarem les posicions equivalents al angle al que apunta cada sensor. Per el sensor làser que apunta endavant no hi ha cap problema, s'afegeix i es crea el servei com amb la càmera.


```

private LaserRangeFinderEntity CreateEpuckIR(int name, int direccionZ)
{
    // Create a Laser Range Finder Entity .
    LaserRangeFinderEntity IR = new LaserRangeFinderEntity(new Pose(
        new Vector3(0.0f, 0.027f, 0.0f),
        TypeConversion.FromXNA(xna.Quaternion.CreateFromAxisAngle(new xna.Vector3(0, 1, 0),
(float)Math.PI)))));
    IR.State.Name = "IR"+name;
    IR.LaserBox.State.DiffuseColor = new Vector4(0.25f, 0.25f, 0.8f, 1.0f);
    IR.LaserBox.BoxState.Dimensions = new Vector3(0.001f, 0.001f, 0.001f);
    // Create LaserRangeFinder simulation service and specify

    // which entity it talks to

    Ifr.Contract.CreateService(
    ConstructorPort,
    Microsoft.Robotics.Simulation.Partners.CreateEntityPartner(
    "http://localhost/" + IR.State.Name));

    return IR;
}

```

Per afegir al robot:

```

    // Create Infra-Red entity front and start simulated Infra-red service
    LaserRangeFinderEntity EpuckIR = CreateEpuckIR(1, 1);
    // insert as child of motor base
    epuckBaseEntity.InsertEntity(EpuckIR);

```

En canvi per poder afegir el làser posterior es necessari crear una classe que derivi de la classe *LaserRangeFinderEntity* modificant la orientació d'aquest de la següent manera:

```

public class MyLaserRangeFinderEntity2 : LaserRangeFinderEntity
{
    public MyLaserRangeFinderEntity2(Pose pose) : base(pose) { }

    public override void Update(FrameUpdate update)
    {
        Pose oldPose = Parent.State.Pose;
        Parent.State.Pose.Orientation = Parent.State.Pose.Orientation *
        Quaternion.FromAxisAngle(0, 1, 0, (float)(Math.PI));
        base.Update(update);
        Parent.State.Pose = oldPose;
    }
}

```

Un cop modificat es crea el servei i s'afegeix al robot de la mateixa manera que el làser frontal. Veure la situació dels làsers a la figura 3.4.

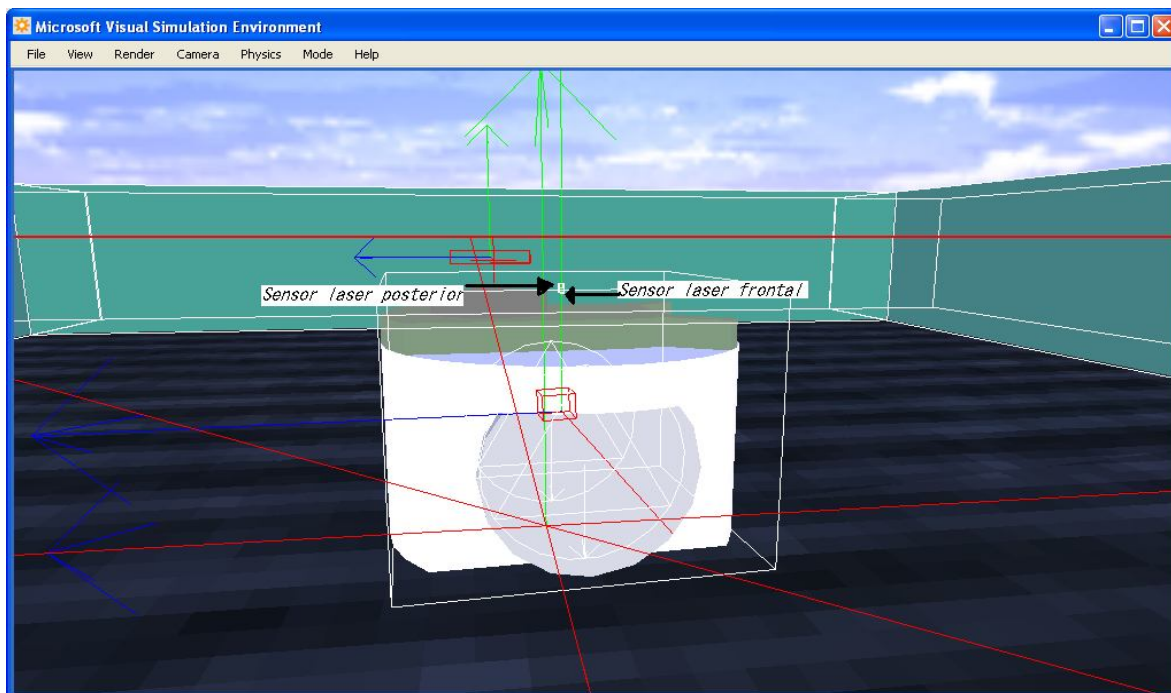


Figura 3.4: Situació del sensors làsers que s'utilitzen per emular els sensors de proximitat

Després d'això ja tenim la simulació acabada.

3.2.7 Modificació del manifest

Els serveis DSS usen un arxiu basat en XML conegut com un manifest per a enumerar els serveis associats a un servei base. Això li indica al mòdul de temps d'execució de MRS la manera que ha de carregar-se el servei de base i interaccionar amb els serveis associats. Encara que el manifest és un arxiu basat en XML, es pot editar amb un editor de text o usar l'editor de DSS Manifest, que s'inclou amb MRS. L'editor de DSS Manifest és una eina visual que permet arrossegar els serveis associats a una superfície de disseny. Abans d'usar l'editor de DSS Manifest, haurà de compilar correctament el servei de base. Per evitar el canvi d'entorn s'ha decidit modificar el fitxer *AutonomEpuck.manifest.xml* directament des del mateix entorn de programació Microsoft Visual C#, ja que l'únic que volem fer és afegir un servei anomenat *SimpleDashboard* (figura 3.5) inclòs dintre de la carpeta `\samples` provinent de la instal·lació de MRS. Aquest servei permet iniciar altres serveis que s'usen per a controlar la funcionalitat específica del robot. Per exemple, el codi del servei *SimEpuck* amb entitats l'escena de la simulació, entre les quals es troba el robot E-puck. No obstant això, no existeix cap codi en el servei per a moure al robot. Aquest codi s'agafa del servei Drive.

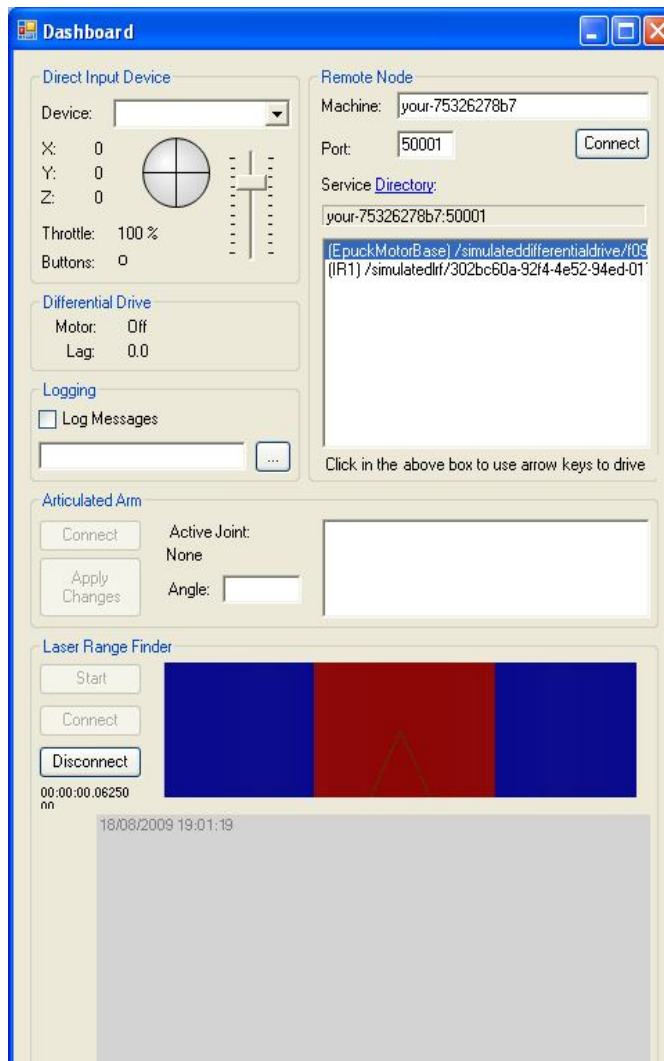


Figura 3.5: Simple Dashboard en funcionament

Per a carregar i usar aquest panell s'ha agregat al manifest afegint aquest codi:

```
<ServiceRecordType>
<dssp:Contract>http://schemas.microsoft.com/robotics/2006/01/simpledashboard.html</dssp:Contract>
</ServiceRecordType>
```

4. Programa de control automàtic sobre la simulació

4.1 Introducció

Microsoft Robotics Developer Studio (RDS) pot implementar el comportament d'un robot escrivint un *servei d'orquestració de serveis*. Els serveis d'orquestració comuniquen amb serveis que realitzen funcions concretes i poden representar la interfície de software o hardware del robot.

En aquest apartat es descriu com s'ha creat un servei d'orquestració que dóna un comportament de desplaçament aleatori per a la simulació de l'apartat anterior d'un robot e-puck equipat amb un dispositiu de mesurament de distància basat en làser, utilitzant la distància en els graus on enfoquen els sensors IR del robot real per tal d'emular-los.

El programa que s'ha decidit fer segueix el següent diagrama (figura 4.1):

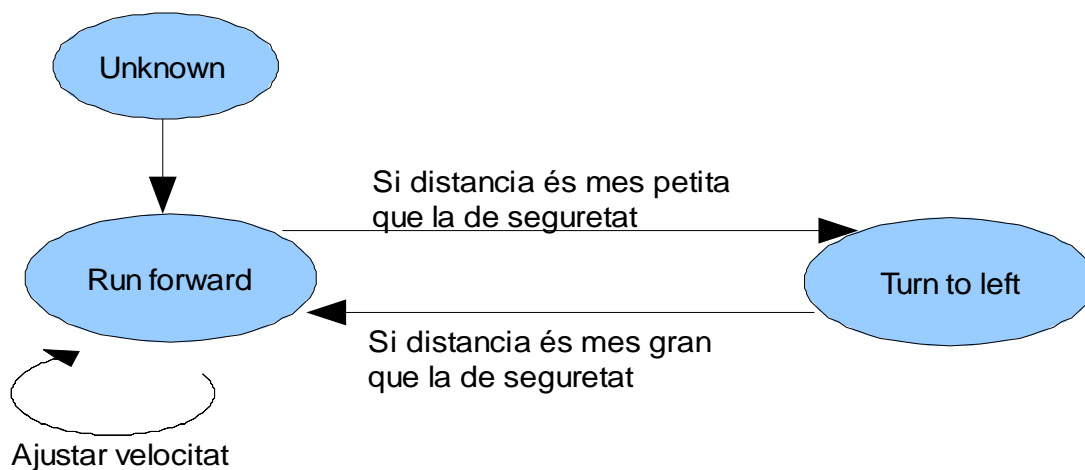


Figura 4.1: Diagrama d'estats del programa de control

El robot avançarà a velocitat màxima fins a acostar-se una mica a la paret, reduirà la velocitat fins arribar a la paret, s'aturarà i girarà cap a l'esquerre fins que la distància a la paret sigui superior a la de seguretat i torni a avançar en línia recta.

Distància és la suma de la distància que detecten els sensors IR0, IR1, IR6, i IR7; dividida entre 4

$$D=(IR0+IR1+IR6+IR7)/4$$

Mes endavant s'explicarà com s'obtenen aquestes distàncies.

4.2 Implementació

Primer s'ha creat el projecte *AutonomEpuck* utilitzant la plantilla *SimpleDssService*.

4.2.1 Components

Per poder implementar aquest comportament primer cal definir els serveis que controlen els sensors i actuadors. En aquest cas s'ha fet servir una implementació del servei abstracte *Drive Contract* per controlar les rodes del e-puck. El *Drive Contract* té les següents operacions: modifica la velocitat de les rodes dreta i esquerra individualment, fa rotar el robot sencera a partir d'un angle donat, desplaça el robot una distància concreta, etc. El servei *Drive* envia una notificació quan el seu estat canvia, incloent-hi la velocitat de rodes.

Un altre servei que s'ha utilitzat és el *SickLRF* que representa un *Laser Range Finder Device* que és un aparell de mesurament de distància: mesura la distància a un obstacle enviant un raig làser i mesurant el temps fins que la seva reflexió es rebí. La majoria dels telèmetres làsers utilitzen un mirall que es fa girar per escanejar el segment d'un arc en comptes d'un punt únic. La precisió del LRF està determinada per la seva resolució espacial (p. ex. nombre de lectures de distància per escanejar) i resolució temporal (escanegi freqüència). Aquesta classe assumeix la configuració següent:

- 180° de escanejat
- 181 lectures de distància per segment (1 grau de resolució)
- 5 escanejades per segon

El servei envia una notificació a cada escanejat. Un escanejada és un conjunt de lectures on cada lectura conté la distància mesurada en un angle específic, distribuïdes en una estructura de taula d'enters. Veure l'equivalència a la figura 4.2.

En el nostre cas només utilitzarem els angles -17, -45, 45,+17 que corresponen als sensors IR0, IR1, IR6, i IR7 respectivament; del robot real.

Aquests angles corresponen a les posicions de la taula 73, 45, 135 i 107 respectivament.

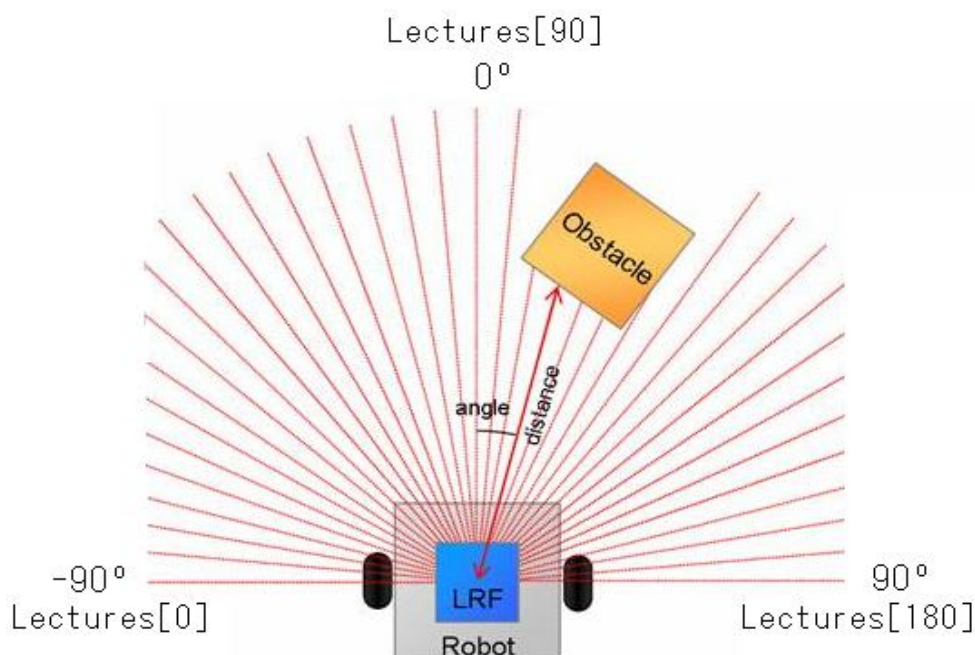


Figura 4.2: Esquema del sensor làser amb angles i equivalència de la taula de lectures

Utilitzant aquest servei, s'ha definit l'orquestració per ser implementada pel servei *AutonomEpuck*. El servei *AutonomEpuck* ha de subscriure el servei *Drive*, i servei de LRF. Quan rep una notificació de qualsevol d'aquests serveis, el servei *AutonomEpuck* ha d'enviar una ordre al servei *Drive* com es veu a la figura 4.3.

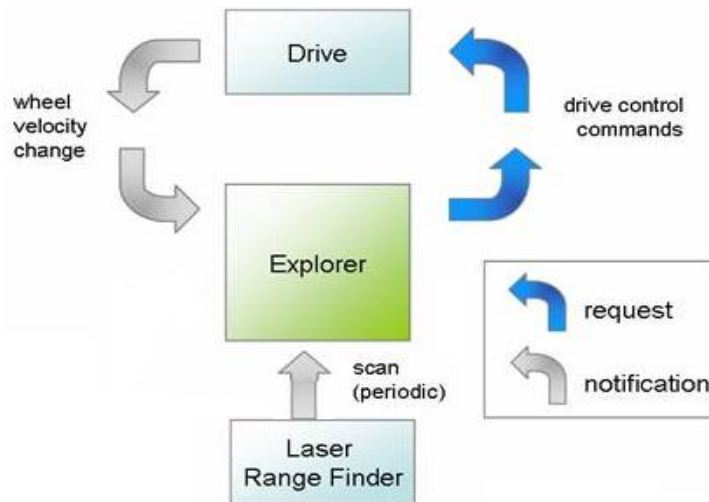


Figura 4.3: Intercanvi d'informació dels serveis al servei *AutonomEpuck*

Per utilitzar aquests serveis, ha d'afegir referències *RoboticsCommon.Proxy.dll* i *SickLRF.Y2005.M12.Proxy.dll*

Després d'afegir les referències, era necessari afegir les següents declaracions de l'espai de noms al fitxer d'implementació de classe *AutonomEpuck.cs*:

```
using drive = Microsoft.Robotics.Services.Drive.Proxy;
using sicklrf = Microsoft.Robotics.Services.Sensors.SickLRF.Proxy;
```

4.2.2 Definir els estats

En la introducció, s'ha definit el comportament desitjat. L'aplicació d'aquest comportament s'ha de conduir per les notificacions diferents dibuixades a la figura 4.3, pensant que qualsevol d'aquestes notificacions pot arribar a qualsevol moment independent de quin estat està el robot. Per exemple, les notificacions de LRF arriben periòdicament però l'acció que resulta clarament depèn tant de si l'explorador se n'està anant endavant com si està girant.

L'enfocament per implementar tal conducta a l'esdeveniment està pensat utilitzant una màquina de tres estats:

Movent-se (Run Forward): El robot té espai lliure endavant.

Girant (Turn to left): El robot gira cap a l'esquerra mentre busca espai lliure endavant.

Desconegut (Unknown): És l'estat inicial. També s'utilitza si es reinicia el làser.

Com que és una màquina d'estats, primer cal definir-los per saber en quin estat es troba el robot. Per fer això s'ha afegit una definició d'enumeració que representa els diferents estats al fitxer *AutonomEpuckState.cs*:

```
[DataContract]
public enum LogicalState
{
    Unknown, Turn, Run
}
```

I s'ha creat la classe *State* per afegir-hi les dades que es compartiran i hi afegim el *LogicalState* que determina l'estat

```
[DataContract]
public class State
{
    private LogicalState _logicalState;

    [DataMember]
    public LogicalState LogicalState
    {
        get { return _logicalState; }
        set { _logicalState = value; }
    }
}
```

Finalment s'han creat funcions per accedir correctament a l'estat:

```
internal bool IsUnknown
{
    get
    {
        return LogicalState != LogicalState.Run ||
            LogicalState != LogicalState.Turn ||
            LogicalState == LogicalState.Unknown;
    }
}

internal bool IsMoving
{
    get
    {
        return LogicalState == LogicalState.Run;
    }
}

internal bool IsTurn
{
    get
    {
        return LogicalState == LogicalState.Turn;
    }
}
```

4.2.3 Definir els Handlers

En els handlers per a les notificacions del làser es pot accedir i canviar l'estat segons la màquina d'estats. Com que totes les notificacions podrien canviar l'estat, el seu handler s'ha d'executar dins de l'*ExclusiveReceiverGroup* de l'activació de serveis.

Per actualitzar l'estat basat en dades làsers, s'ha afegit una petició de *LaserRangeFinderUpdate* als seus tipus al fitxer *AutonomEpuckTypes.cs*:

```
class LaserRangeFinderUpdate : Update<sicklrf.State, PortSet<DefaultUpdateResponseType, Fault>>
{
    public LaserRangeFinderUpdate(sicklrf.State body)
        : base(body)
    {}

    public LaserRangeFinderUpdate()
    {}
}
```

També s'ha afegit el *DriveUpdate* per les dades de les rodes.

```
public class DriveUpdate : Update<drive.DriveDifferentialTwoWheelState,
PortSet<DefaultUpdateResponseType, Fault>>
{
    public DriveUpdate(drive.DriveDifferentialTwoWheelState body)
        : base(body)
    {}

    public DriveUpdate()
    {}
}
```

I s'han afegit al port d'operacions

```
class AutonomEpuckOperations : PortSet<
    DsspDefaultLookup,
    DsspDefaultDrop,
    Get,
    DriveUpdate,
    LaserRangeFinderUpdate>
{
}
```

A continuació s'han implementat les transicions provocades per el làser en el handler *LaserRangeFinderUpdate*

```
/// <summary>
/// Handles the <typeparamref name="LaserRangeFinderUpdate"/> request.
/// </summary>
/// <param name="update">request</param>
void LaserRangeFinderUpdateHandler(LaserRangeFinderUpdate update)
{
    sickIrf.State laserData = update.Body;

    _state.MostRecentLaser = laserData.TimeStamp;

    int distance = (IR0(laserData) + IR1(laserData) + IR6(laserData) + IR7(laserData)) / 4;
    // AvoidCollision and EnterOpenSpace have precedence over
    // all other state transitions and are thus handled first.
    AvoidCollision(distance, laserData);

    update.ResponsePort.Post(DefaultUpdateResponseType.Instance);
}
```

El qual s'executarà cada cop que es realitzi una actualització del làser.

Aquí s'actualitza la distància que s'ha explicat avanç i s'executa el mètode *AvoidCollision*

```
private void AvoidCollision(int distance, sickIrf.State laserData)
{
    if (distance < ObstacleDistance && _state.IsMoving)
    {
        StopMoving();
        _state.LogicalState = LogicalState.Turn;
        TurnEpuck();
    }
    else
    {
        _state.LogicalState = LogicalState.Run;
        AdjustVelocity(laserData, distance);
    }
}
```


Aquest mètode comprova si la distància entre el robot i l'obstacle és prou gran per continuar endavant, en aquest cas es realitzarà un ajust de la velocitat segons la distància al obstacle i es posarà en l'estat *Run*. En el cas d'estar molt proper a l'objecte, el robot s'aturarà, es posarà en l'estat *Turn* i començarà a girar a l'esquerra fins que en una següent execució d'aquest mètode la distància sigui superior a la de seguretat i torni a anar endavant.

```
private void AdjustVelocity(sickIrf.State laserData, int distance)
{
    if (distance > FreeDistance)
    {
        MoveForward(MaximumForwardVelocity);
    }
    else if (distance > AwareOfObstacleDistance)
    {
        MoveForward(MaximumForwardVelocity / 2);
    }
    else
    {
        MoveForward(MaximumForwardVelocity / 4);
    }
}
```

Per tal de controlar els motors s'han afegit els Handlers: *DriveUpdateNotification*, *DriveUpdateHandler*, *DriveStateUpdate* que actualitzen l'estat del *Drive*.

```
/// <summary>
/// Handles Update notification from the Drive partner
/// </summary>
/// <remarks>Posts a <typeparamref name="DriveUpdate"/> request to itself.</remarks>
/// <param name="update">notification</param>
void DriveUpdateNotification(drive.Update update)
{
    _mainPort.Post(new DriveUpdate(update.Body));
}

/// <summary>
/// Handles DriveUpdate request
/// </summary>
/// <param name="update">request</param>
void DriveUpdateHandler(DriveUpdate update)
{
    DriveStateUpdate(update.Body);
    update.ResponsePort.Post(DefaultUpdateResponseType.Instance);
}

// Update our internal state using a drive state
void DriveStateUpdate(drive.DriveDifferentialTwoWheelState s)
{
    _state.DriveState = s;
}
```

4.2.4. Mètodes auxiliars per el servei Drive

S'han afegit uns mètodes auxiliars per tal de controlar les rodes utilitzant el servei *Drive*:

Aquest mètode activa els motors i envia una petició de modificació de la velocitat de les rodes, ambdós a la mateixa velocitat que indica el paràmetre d'entrada.

```
private PortSet<DefaultUpdateResponseType, Fault> MoveForward(int speed)
{
    if ((_state.DriveState == null || !_state.DriveState.IsEnabled) && speed != 0)
    {
        EnableMotor();
    }
    else
    {
        DisableMotor();
    }

    drive.SetDriveSpeedRequest request = new drive.SetDriveSpeedRequest();

    request.LeftWheelSpeed = (double)speed / 1000.0; // millimeters to meters
    request.RightWheelSpeed = (double)speed / 1000.0; // millimeters to meters

    return _drivePort.SetDriveSpeed(request);
}
```

Aquest segon mètode envia la petició de modificació de la velocitat de les rodes, a la mateixa velocitat però en sentits oposats, d'aquesta manera el robot gira sobre si mateix.

```
private PortSet<DefaultUpdateResponseType, Fault> TurnEpuck()
{
    if ((_state.DriveState == null || !_state.DriveState.IsEnabled))
    {
        EnableMotor();
    }
    else
    {
        DisableMotor();
    }

    drive.SetDriveSpeedRequest request = new drive.SetDriveSpeedRequest();

    request.LeftWheelSpeed = (double)0.2; // millimeters to meters
    request.RightWheelSpeed = -(double)0.2; // millimeters to meters

    return _drivePort.SetDriveSpeed(request);
}
```

Aquest mètode simplement crida al mètode *MoveForward* amb velocitat 0:

```
PortSet<DefaultUpdateResponseType, Fault> StopMoving()
{
    return MoveForward(0);
}
```

Els següents mètodes serveixen per activar o desactivar el motor mitjançant la petició corresponent.

```
PortSet<DefaultUpdateResponseType, Fault> EnableMotor(bool enable)
{
    drive.EnableDriveRequest request = new drive.EnableDriveRequest();
    request.Enable = enable;

    return _drivePort.EnableDrive(request);
}
```

```
PortSet<DefaultUpdateResponseType, Fault> EnableMotor()
{
    return EnableMotor(true);
}
```

```
PortSet<DefaultUpdateResponseType, Fault> DisableMotor()
{
    return EnableMotor(false);
}
```

4.2.5. Ignorar antigues notifikacions

Molts serveis envien notifikacions periòdicament. Les notifikacions fan cua en el port fins que se'ls dona servei. Això significa que la informació potencialment antiquada es guardi innecessàriament al voltant i finalment se li doni servei. Si el donar servei al missatge és computacionalment car o els missatges són grans, això és indesitjable. Fins i tot pitjor, les notifikacions poden arribar més ràpid que els donar-hi servei i s'acumularan.

Així, en algunes situacions és necessari només donar servei la notifikació més recent i saltar aquells que es rebin abans. En el nostre cas, les notifikacions des del LRF tenen aquest problema, que cap sentit no analitzi dades velles quan les dades noves estan disponible.

Per solucionar aquest problema s'ha implementat el Handler *LaserReplaceNotification* de manera que s'obtingui únicament l'actualització més recent utilitzant la funció *GetMostRecentLaserNotification* que s'explicarà més endavant. En acabat cal reactivar el handler per tal d'esperar noves actualitzacions.

```
IEnumerator<ITask> LaserReplaceNotification(sicklrf.Replace replace)
{
    // When this handler is called a couple of notifications may
    // have piled up. We only want the most recent one.
    sicklrf.State laserData = GetMostRecentLaserNotification(replace.Body);

    LaserRangeFinderUpdate request = new LaserRangeFinderUpdate(laserData);

    _mainPort.Post(request);

    yield return Arbiter.Choice(
        request.ResponsePort,
        delegate(DefaultUpdateResponseType response) {},
        delegate(Fault fault) {}
    );

    // Skip messages that have been queued up in the meantime.
    // The notification that are lingering are out of data by now.
}
```

```

    GetMostRecentLaserNotification(laserData);

    // Reactivate the handler.
    Activate(
        Arbiter.ReceiveWithIterator<sickIrf.Replace>(false, _laserNotify, LaserReplaceNotification)
    );
}

```

En la funció *GetMostRecentLaserNotification* s'ha utilitzat el mètode *Test()*, que retorna el pròxim missatge, per llegit tots els missatges i buscar el més recent i retornar-lo. El mètode *Test()* només treballa amb ports que no s'activen en mode persistent per això després de treballar amb el missatge s'ha de reactivar el Handler anterior.

```

private sickIrf.State GetMostRecentLaserNotification(sickIrf.State laserData)
{
    sickIrf.Replace testReplace;

    // _laserNotify is a PortSet<>, P3 represents IPort<sickIrf.Replace> that
    // the portset contains
    int count = _laserNotify.P3.ItemCount - 1;

    for (int i = 0; i < count; i++)
    {
        testReplace = _laserNotify.Test<sickIrf.Replace>();
        if (testReplace.Body.TimeStamp > laserData.TimeStamp)
        {
            laserData = testReplace.Body;
        }
    }
    return laserData;
}

```

4.2.6. Modificació del mètode Start()

En el mètode *Start* primer s'ha definit l'estat i s'ha posat l'activació de notificació dels diferents Handlers de forma que els Handlers *LaserRangeFinderUpdateHandler* i *DriveUpdateHandler* s'atenguin exclusivament per protegir les dades. Finalment s'activa la recepció de missatges del làser de forma iterativa i es subscriuen les notificacions del *Drive* i del Làser als ports corresponents.

```

protected override void Start()
{
    if (_state == null)
        CreateState();

    Activate(
        Arbiter.Interleave(
            new TeardownReceiverGroup(
                Arbiter.Receive<DsspDefaultDrop>(false, _mainPort, DropHandler)
            ),
            new ExclusiveReceiverGroup(
                Arbiter.Receive<LaserRangeFinderUpdate>(true, _mainPort,
                LaserRangeFinderUpdateHandler),
                Arbiter.Receive<DriveUpdate>(true, _mainPort, DriveUpdateHandler)
            ),
            new ConcurrentReceiverGroup(
                Arbiter.Receive<Get>(true, _mainPort, GetHandler),
                Arbiter.Receive<dssp.DsspDefaultLookup>(true, _mainPort, DefaultLookupHandler)
            )
        )
    )
}

```

```

);

    Activate(
    Arbiter.Interleave(
        new TeardownReceiverGroup(),
        new ExclusiveReceiverGroup(),
        new ConcurrentReceiverGroup(
            Arbiter.Receive<drive.Update>(true, _driveNotify, DriveUpdateNotification)
        )
    )
);

    Activate(
        Arbiter.ReceiveWithIterator<sicklrf.Replace>(false, _laserNotify, LaserReplaceNotification)
    );

    _drivePort.Subscribe(_driveNotify);
    _laserPort.Subscribe(_laserNotify);
}

```

4.2.7. Modificació del manifest

Finalment s'ha modificat el manifest per tal d'associar aquest servei d'orquestració amb el servei de simulació del robot e-puck de l'apartat anterior.

Per fer això s'ha afegit al fitxer *AutonomEpuck.manifest.xml* el següent codi:

```

<ServiceRecordType>
    <dssp:Contract>http://schemas.tempuri.org/2009/01/simepuck.html</dssp:Contract>
</ServiceRecordType>

```

D'aquesta manera al carregar el manifest s'obriran els dos serveis i es comunicaran entre ells per controlar el robot dintre l'entorn de simulació.

5. Interfície de connexió i comunicació entre el robot e-puck i MRS

5.1 Introducció

Un cop s'ha treballat amb simulacions el que es vol és treballar sobre el robot real per tal de poder afrontar problemes i situacions reals. Per fer això primer s'ha de conèixer com es comunica el robot e-puck i quin protocol fa servir. Després de llegir la documentació del robot e-puck es va trobar el programa principal que executa el robot e-puck al iniciar-se:

```
int main() {
    int selector;

    //system initialization
    e_init_port(); // configure port pins
    e_start_agendas_processing();
    e_init_motors();
    e_init_uart1(); // initialize UART to 115200 Kbaud
    e_init_ad_scan();

    //Reset if Power on (some problem for few robots)
    if (RCONbits.POR) {
        RCONbits.POR=0;
        __asm__ volatile ("reset");
    }
    // Decide upon program
    selector=getselector();
    sprintf(buffer, "Selector pos %d\r\n", selector);
    e_send_uart1_char(buffer, strlen(buffer));
    if (selector==0) {
        run_accelerometer();
    } else if (selector==1) {
        run_locatesound();
    } else if (selector==2) {
        run_wallfollow();
    } else if (selector==3) {
        run_asercom(); // advanced sercom protocol
    } else if (selector==5) { // sensor "feedback display"
        run_SensDispl();
    } else if (selector==6) { // camera point to light
        run_CameraTurn();
    } else if (selector==7) { // act like the ASL RS232 - I2C translator
        run_translatorI2C();
    } else {
        run_breitenberg();
    }
    while(1);
    return 0;
}
```

Després de inicialitzar i configurar l'epuck, s'executa un programa depenen de la posició del selector. El programa que ens interessa és el de la posició 3, que és un programa que s'utilitza per comunicar el robot amb un ordinador utilitzant el port bluetooth a partir del UART. Aquest programa es troba a l'arxiu *Asercom.c*.

Per fer això llegeix caràcters contínuament fins a trobar la marca de final d'instrucció '\n' o '\r'. Un cop té la instrucció mirarà què demana segons el primer caràcter i li oferirà el servei adient amb la resposta corresponent utilitzant el protocol que s'ha explicat al apartat 2.3.4 d'aquesta memòria.

Un cop es va comprendre com comunicar-se amb el robot e-puck es va buscar com accedir als ports i es va trobar la llibreria *System.IO.Ports* que conté les classes per controlar els ports sèrie. La classe més important és *SerialPort* que proporciona una estructura per entrades i sortides síncrones, i accés a les propietats del port sèrie. Finalment es va decidir que utilitzant aquesta llibreria no calia crear un projecte per accedir al robot sinó que amb una sola classe es podria implementar aquest servei.

5.2 Implementació

5.2.1 Afegir llibreries necessàries i instància del port.

Per poder utilitzar les llibreries s'han d'afegit les següents declaracions:

```
using System.IO.Ports;  
using System.Threading;
```

El següent que s'ha fet és declarar una instància de la classe *SerialPort* el qual representarà el port sèrie al qual s'associarà al dispositiu bluetooth. El declarem privat perquè no s'hi accedeixi externament.

```
private SerialPort _epuckPort = null;
```

5.2.2 Connexió amb el robot.

Per connectar amb el robot e-puck és fa servir la instància del tipus *SerialPort* però primer s'ha d'inicialitzar amb els paràmetres adequadament, després de llegir el codi intern del robot e-puck s'ha descobert que treballa a una velocitat de 115200 bauds, sense paritat, amb 8 bits de dades i un de parada. El codi que inicialitza el port és el següent:

```
_epuckPort = new SerialPort("COM5", 115200, Parity.None, 8, StopBits.One);
```

I per obrir la connexió s'utilitza la següent funció:

```
_epuckPort.Open();
```

5.2.3. Mètodes Set

Per poder modificar els actuadors del robot e-puck s'han creat una serie de mètodes que modifiquen l'estat dels actuadors: Leds, motors, so de l'altaveu i paràmetres de la càmera. Per fer això cada mètode segueix el següent algoritme:

1. Comprovació dels paràmetres d'entrada. Aquest pas pot no ser necessari.
2. Netejar Buffer d'entrada.
3. Escriure al port la instrucció necessària amb el codi i els paràmetres segons el protocol explicat anteriorment.
4. Llegir el que retorna el robot i retornar-ho.

Per exemple:

```
public string SetSpeed(int vleft, int vright)  
{  
    ClearBuffer();  
    _epuckPort.Write("D," + vleft.ToString() + "," + vright.ToString() + "\r");  
    return LlegirEntrada();  
}
```

El que retorna el robot pot ser utilitzat com a detector d'error ja que retorna el codi d'instrucció en minúscula.

5.2.4. Mètodes Get

Per accedir als valors que recullen els sensors del robot e-puck s'han creat uns altres mètodes que demanen al robot que li envii el que recullen els sensors que necessita. Per cada un dels possibles tipus de sensor s'ha creat un mètode diferent de manera que els sensors d'un mateix tipus es demanen amb una mateixa instrucció i el robot retorna un *string* amb el valor que tenen tots ells. Aquests mètodes segueixen el següent algoritme:

1. Escriure al port d'instrucció necessària amb el codi del sensor que es demana segons el protocol.
2. Llegir el que retorna el robot i retornar-ho.

Per exemple:

```
public string GetMotorPosition()
{
    _epuckPort.Write("Q\r");
    return LlegirEntrada();
}
```

Com que no sempre interessa tenir un string amb tots els valors del sensors i algunes vegades és més interessant accedir a un sol paràmetre i obtenir l'enter que indica el sensor, s'han creat uns altres mètodes que amb un paràmetre d'entrada es determina quin dels sensors interessa i mitjançant una cerca dintre del string que retorna el robot seguint el següent algoritme:

1. Cridar el mètode corresponent que retorna el string que respon el robot.
2. Comprovar si el string es correcte, mirant si correspon el primer caràcter amb el codi, si a continuació hi ha una coma i si el paràmetre d'entrada es correcte.
3. Fer (numero de vegades que equival al numero del paràmetre desitjat).
 1. Mentre no marca de fi.
 1. Llegir caràcter i guarda en una cadena.
 2. Convertir cadena de caràcters numèrics en enter.
4. Retornar l'últim enter convertit que és el valor del sensor desitjat.

Per exemple:

```
public int GetMotorPosition(int num)
{
    string s = GetMotorPosition();
    if (s.Length > 5 && s[0] == 'q' && s[1] == ',' && num >= 0 && num <= 1)
    {
        char c;
        int j = 1;
        string a;
        int x = 1000;
        for (int i = 0; i <= num; i++)
        {
            j++;
            c = s[j];
            a = "";
            while (c != ',' && c != '\r') {
                a += c;
                j++;
                c = s[j];
            }
            int.TryParse(a, out x);
        }
        return x; }
}
```


5.2.5. Altres mètodes

El robot e-puck també pot atendre unes altres peticions que no modifiquen els actuadors ni demanen els valors dels sensors. Aquestes peticions són les de calibrar els sensors, fer un reset al robot i aturar-lo. Per fer aquestes peticions al robot e-puck es segueix el següent algoritme:

- 1 Escriure al port la instrucció necessària amb el codi corresponent a la petició segons el protocol explicat anteriorment.
- 2 Llegir el que retorna el robot i retornar-ho per tal de que l'usuari pugui comprovar si ha arribat a complir-se la petició.

5.2.6 Mètodes auxiliars del port.

Per tal de poder accedir a l'informació del port s'han creat dues funcions:

- *LlegirEntrada* que retorna el string enviat per el robot e-puck que es rep al port com a resposta a qualsevol petició que se li a fet.
- *ClearBuffer* que llegeix tots els bytes que estan a la cua del port, de manera que queden eliminats de la cua.

5.2.7. Desconnexió del robot.

Per desconnectar el robot e-puck tan sols s'ha de tancar el port, però abans de fer-ho cal assegurar que està aturat, els motors, els leds i l'altaveu. El codi resultant és el següent:

```
public void Disconnect()
{
    _epuckPort.Write("S\r");
    _epuckPort.Write("T,0\r");
    _epuckPort.Close();
}
```

5.3. Utilització de la classe

Per poder utilitzar aquesta classe tan sols cal seguir els següents passos:

- Copiar l'arxiu *EpuckComm.cs* a la carpeta del projecte on es vol utilitzar.
- A la plataforma de programació fer Proyecto>Agregar Elemento existente o fer Mayús+Alt+A. I afegir l'arxiu *EpuckComm.cs*. Veure figura 5.1.
- Finalment obrir l'arxiu i canviar la declaració *namespace EpuckBleTooth* per *namespace projecte*. On projecte ha de ser el nom del projecte actual.

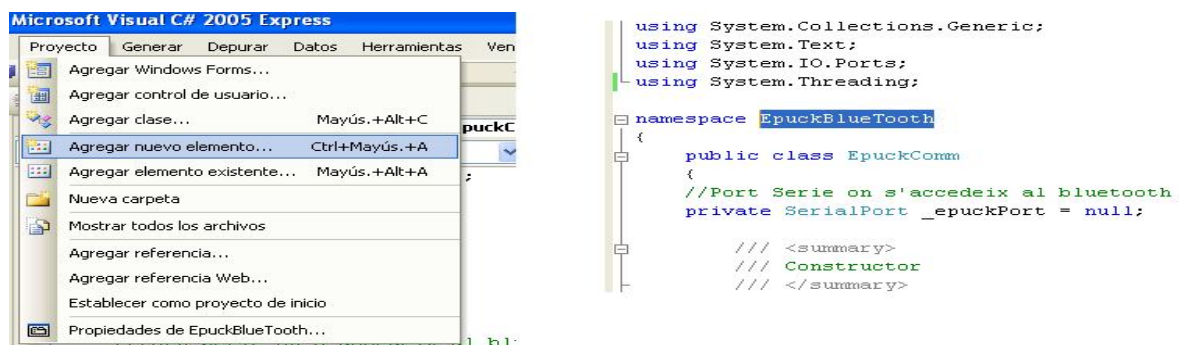


Figura 5.1: Utilització de la classe. A l'esquerra agregar element, a la dreta canvia declaració namespace

6. Aplicació gràfica de monitorització i control del e-puck

6.1 Introducció

Un cop tenim la possibilitat de connectar i comunicar ordres i peticions al robot e-puck real, s'ha decidit crear una interfície gràfica d'usuari d'una aplicació. En aquesta interfície es podran modificar els actuadors i obtenir el valor dels sensors a distància tan sols prement un botó.

Aquesta interfície estarà formada per diferents botons i quadres de text, on cada botó tindrà una funcionalitat determinada. Uns serviran per modificar l'estat dels actuadors i d'altres serviran per obtenir valors dels sensors del robot real i retornar-los en un quadre de text de la mateixa plataforma.

6.2 Disseny i Implementació

Per implementar aquesta interfície d'usuari s'han realitzat els següents passos:

6.2.1. Crear el projecte i afegir els elements necessaris

El primer que es va fer va ser crear un nou projecte que s'ha anomenat *EpuckBlueTooth* utilitzant la plantilla *Aplicación para Windows* de les plantilles instal·lades de Visual Studio tal com es veu en la Figura 6.1.

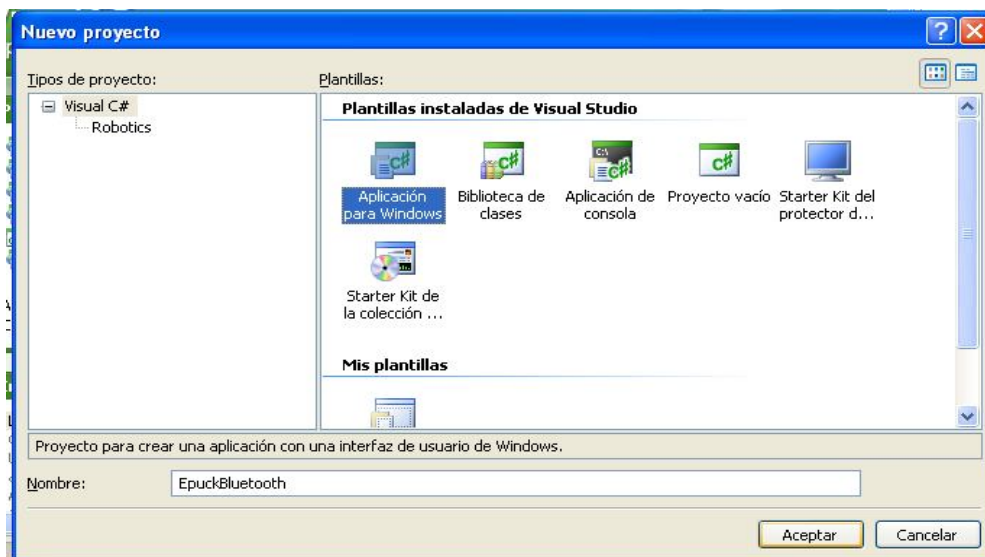


Figura 6.1: Creació del projecte Bluetooth

Això crearà un projecte amb un fitxer de classe Program.cs on automàticament s'escriu el codi necessari per executar la interfície, aquest codi no es modificarà; i una classe de tipus *Form* composta d'un quadre de disseny d'interfície on es crearà la part visual de la plataforma amb botons i quadres de text; i d'un fitxer de classe on s'implementarà els mètodes que s'activaran quan hi hagi una acció en la interfície d'usuari, únicament seran de tipus clic sobre els botons.

Per tal de poder accedir al robot e-puck s'utilitza la classe creada en el capítol 5. Per poder utilitzar-la s'ha afegit al projecte tal com s'ha explicat a l'apartat 5.3. A continuació s'ha creat una instància de la classe *EpuckComm*:

```
public partial class Form1 : Form
{
    EpuckComm epuck = null;
```

I en el constructor de *Form1()* s'hi ha afegit *epuck = new EpuckComm();* per accedir-hi.

6.2.2. Creació dels botons de control de motors

Primerament s'han creat cinc botons per donar les ordres d'avançar, retrocedir, girar a l'esquerre, girar a la dreta i parar als motors. Per afegir aquests botons s'ha d'obrir la vista de disseny del *Form1.cs* i del quadre d'eines, que es troba a l'esquerre del entorn de treball tal com s'indica a la figura 6.2, s'ha afegit els cinc elements de tipus *Button*.

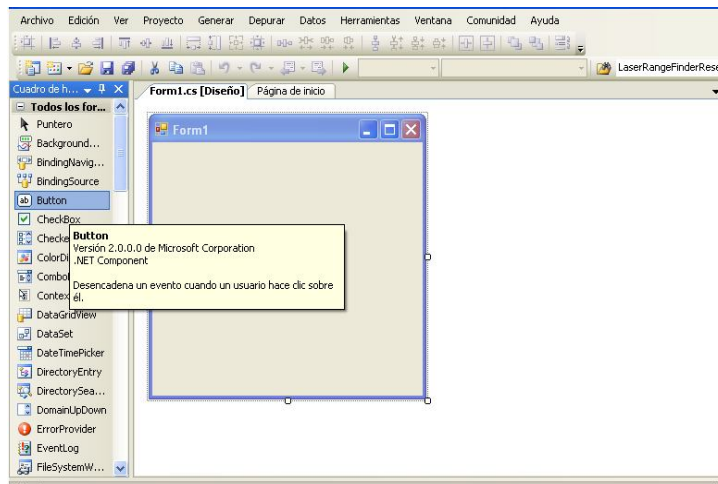


Figura 6.2: Creació del botons

Un cop afegits els botons s'han canviat les propietats d'aquests de la següent manera:

- Button1: **Nom:** Endavant; **Tamany:** 35,23; **Text:** 5;
- Button2: **Nom:** esquerre; **Tamany:** 35,23; **Text:** 3;
- Button3: **Nom:** dreta; **Tamany:** 35,23; **Text:** 4;
- Button4: **Nom:** enredera; **Tamany:** 35,23; **Text:** 6;
- Button5: **Nom:** stop; **Tamany:** 35,23; **Text:** 1;
- A tots ells: **Font:** Marlett **Tamany:** 9.75pt

Després de canviar les propietats s'han canviat les posicions perquè quedes com es veu a la figura 6.3.

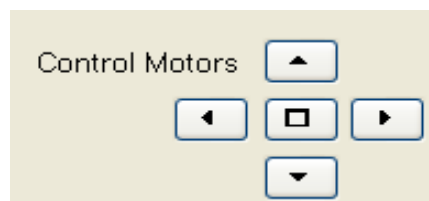


Figura 6.3: Disposició dels botons de control dels motors

6.2.3. Implementació dels mètodes de control dels motors

Per controlar els motors s'han d'implementar els mètodes que s'executaran al fer clic amb el ratolí sobre els botons, per fer això el *Visual C#* permet fer clic sobre el botó a la pantalla de disseny i automàticament crea el mètode que s'executarà al fer el clic.

A continuació a cada un d'ells s'hi ha posat una crida al mètode *SetSpeed* de la classe *EpuckComm*, amb les següents velocitats motor esquerre, motor dret:

Endavant: 500,500 Enderrera: -500,-500 Dreta: 500,-500 Esquerre:-500,500 Stop: 0,0

6.2.4. Creació dels botons de control de tots els leds

Primerament s'han creat vuit botons per modificar l'estat dels leds vermells de l'anell exterior de l'epuck, a cada un se li ha canviat el nom per Ledn on n es el numero de led i s'hi ha posat el text Ln de la mateixa manera. També s'han canviat les dimensions a 28,20

A continuació s'han creat tres botons més, un per el led central, un altre per el led frontal mes potent i finalment un botó per controlar tot l'anell exterior de leds. Després s'ha canviat el nom i el text:

- Led Central: **Nom**: buttonFrontLed **Text**: Front
- Led del cos: **Nom**: BodyLed **Text**: Body
- Anell de leds: **Nom**: Tots **Text**: Tots

I s'han canviat les dimensions a 39,20.

Un cop creats els botons s'han ordenat de manera que es vegi esquemàticament la situació dels leds en el robot com es pot veure a la figura 6.4.

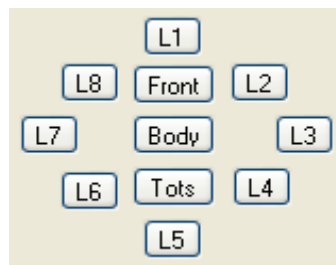


Figura 6.4: Distribució dels botons que controlen els Leds

6.2.5. Implementació dels mètodes de control dels leds

Per modificar l'estat dels leds s'han creat els mètodes que s'executen al fe clic sobre els botons de la mateixa manera que en l'apartat 6.2.3. A continuació s'ha utilitzat el mètode *SetLed* de la classe *EpuckComm* per els leds de l'anell exterior amb els paràmetres numero de led, 2. On el 2 indica inversió de l'estat actual.

Per el mètode del botó Front i Body s'han utilitzat els mètodes *SetFrontLed* i *SetBodyLed* respectivament de la classe *EpuckComm* amb paràmetre d'entrada igual a 2 per indicar que es vol invertir l'estat actual.

Per el mètode del botó *Tots* s'ha utilitzat el mètode *SetLed* però posant com a nombre de led el nombre vuit per indicar que es vol modificar tots els Leds. Però després de provar el funcionament d'aquest botó es va veure que no funcionava sempre si es posava el 2 que indica que es vol fer un canvi d'estat, per aquest fet es va decidir que s'havia de fer un control de l'estat general dels leds mitjançant un camp privat de tipus enter amb tres possibles valors 0 tots apagats, 1 tots encesos, 2 indeterminat. I afegir a tots els mètodes dels leds de l'anell la següent instrucció:

```
allLeds = 2;
```

I en el mètode del botó *Tots* si tots els Leds estan apagats els encén i viceversa i si estan en estat indeterminat els apaga.

6.2.6 Disseny dels elements d'obtenció de dades.

Per poder tenir accés als sensors del robot s'ha afegit un botó per cada tipus de sensor. També s'ha afegit un botó per obtenir el text d'ajuda del robot i un altre per obtenir la versió del controlador. A continuació es llista els botons amb les seves propietats:

· <u>Sensors IR</u>	Nom: IR	Text: IR
· <u>Accelerometres</u>	Nom: Accelerometre	Text: Accelerometre
· <u>Posició dels motors</u>	Nom: PosMotor	Text: Posicio Motor
· <u>Sensors de llum</u>	Nom: SensoLlum	Text: Sensor Llum
· <u>Micròfons</u>	Nom: microfon	Text: Microfon
· <u>Obtenir l'ajuda:</u>	Nom: Help	Text: Help
· <u>Obtenir versió:</u>	Nom: Versio	Text: Versio

Un cop tenim els botons creats es va decidir que per mostrar la informació rebuda es faria mitjançant un *TextBox* i tot es mostraria aquí. Aquest s'anomena *textBox2*. Però mes endavant es va decidir que només s'utilitzaria per mostrar l'ajuda, la versió, els micròfons i els sensors de llum tal com son enviats per el robot e-puck. I per mostrar el resultat dels sensors acceleròmetres, IR i posició dels motors es crearia un *textbox* per a cadascun dels sensors de manera que es poguessin mostrar contínuament i de manera mes gràfica. A tots aquests *textBox* se'ls canviarà la propietat *ReadOnly* a *True*.

Per tant es van crear tres *textbox*; *AccelX*, *AccelY* i *AccelZ* per representar l'acceleració als eixos x,y i z respectivament i es van distribuir com es veu a la figura 6.5.



Figura 6.5: Distribució dels elements per obtenir les acceleracions

Vuit *textBox* IR0...IR7 per representar els 8 sensors IR que es van distribuir al voltant dels botons dels leds situats més o menys a la posició corresponen on es troben els sensors al robot real tal com es veu a la figura 6.6.

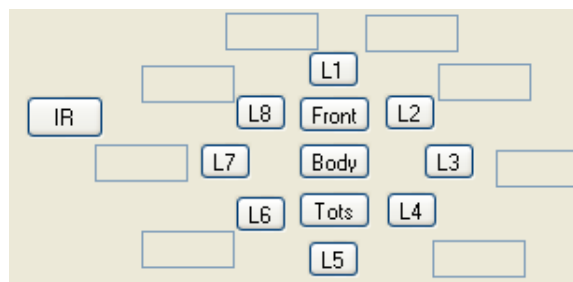


Figura 6.6: Distribució dels elements per mostrar les lectures dels sensors IR

Finalment s'han afegit dos *textBox* més per representar a quina posició es troben els motors. I s'han distribuït com es mostra en la figura 6.7.



Figura 6.7: Distribució dels elements necessaris per conèixer la posició dels motors

6.2.7. Implementació dels mètodes d'obtenció de dades.

Primerament s'han creat tots els mètodes fent doble clic sobre els botons. A continuació s'han implementat els mètodes dels botons *Help*, *Versio*, *Microfon* i *Sensor Ilum* que són molt similars, cadascun crida un dels mètodes de *EpuckComm*: *GetHelp()*, *GetVersion()*, *GetMicrophone()* i *GetAmbientLightSensors()* respectivament. I a continuació després de comprovar si s'ha rebut la resposta correcte es mostra en el *textBox2*.

Tot seguit s'han implementat el mètodes de l'acceleròmetre, els sensors IR i la posició dels motors que també són similars. Com que s'ha decidit que aquests sensors s'actualitzessin constantment s'han de crear subprocessos que s'executin amb un per sempre mitjançant un *while(true)* fins que el pare els mati.

Per fer això el pare ha de controlar si s'ha premut el botó, per fer-ho s'ha utilitzat un camp privat de tipus booleà per a cada pare que si és el primer cop que es fa clic crea un *Thread* prèviament definit i se li assigna el mètode que executarà, a continuació s'activa; quan es torna a prémer el botó es mata aquest *Thread*, per exemple es pot veure en el mètode del botó de l'acceleròmetre:

```
private void Accelerometre_Click_1(object sender, EventArgs e)
{
    if (accel == false)
    {
        accel = true;
        this.a = new Thread(new ThreadStart(this.RutinAccelerometre));
        a.Start();
    }
    else
    {
        a.Abort();
        accel = false;
    }
}
```

On *RutinAccelerometre* és el mètode que s'executarà. Aquest mètode *while(true)* estarà demanant al robot les dades de l'acceleròmetre i realitzant un tractament del *string* rebut que mostra l'acceleració dels eixos x,y i z en els *textBox* corresponents. Aquest tractament consisteix en passar els separadors dels paràmetres, guardar els caràcters en un nou *string* fins el següent separador i convertir aquest nou *string* a un *int*. Tot aquest procés de conversió no és necessari ja que els *textBox* s'hi mostren *strings* però s'ha fet perquè es vegi com obtenir els valors dels sensors quan sigui necessari.

La rutina associada al botó dels sensors IR és similar a l'anterior però cridant al mètode *RutinAccelerometre()* de *EpuckComm* i mostrant els valors dels vuit IR en els *textBox* que hem creat per aquesta finalitat.

Finalment la rutina associada al botó de la posició del motor crida dues vegades el mètode *GetMotorPosition* però amb els paràmetres d'entrada 0 per el motor esquerre i 1 per el motor dret i es mostren els resultats en el *textBox* assignat.

Al intentar mostrar en els *textBox* directament es va veure que no s'hi pot accedir des del subprocés diferent al que l'ha creat, per això s'han creat uns mètodes que permeten l'accés segur als diferents *textBox*.

6.2.8. Disseny dels elements necessaris per reproduir un so.

Per poder reproduir un dels sons programats al robot e-puck hem creat un botó per activar la reproducció del so amb el nom *soroll* i el text *So n°*; i un *textBox* d'entrada anomenat *textBoxSo* per introduir-hi un enter entre 1 i 5 per seleccionar quin dels cinc sons es vol reproduir o un nombre major per aturar l'altaveu. Els elements s'han distribuït com es veu a la figura 6.8.



Figura 6.8: Distribució dels elements usats per reproduir un so

6.2.9. Implementació del mètode per reproduir un so a l'altaveu.

Per reproduir un so es crida el mètode *PlaySound* de *EpuckComm* amb el paràmetre d'entrada el numero introduït al *textBoxSo* com es veu en el següent codi:

```
private void Soroll_Click(object sender, EventArgs e)
{
    epuck.PlaySound(int.Parse(this.textBoxSo.Text.Trim()));
}
```

6.2.10. Creació i implementació del botó Borrar

Per poder esborrar el que hi ha dintre del *textBox2* on s'escriuen l'ajuda, la versió, el valor dels micròfons i dels sensors de llum s'ha creat un botó que netegi aquest. S'ha col·locat el botó com es veu en la figura 6.9.

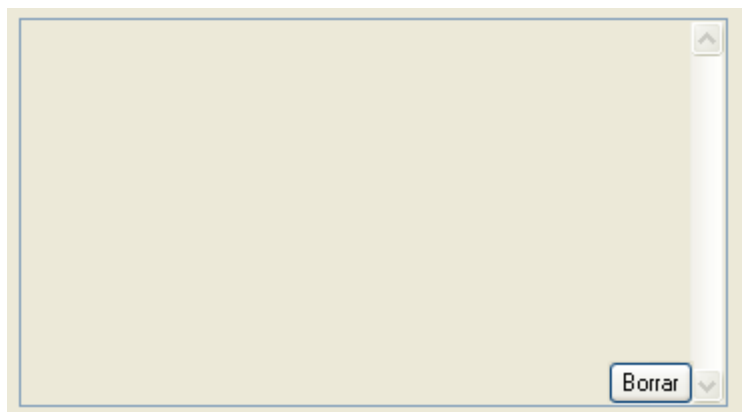


Figura 6.9: TextBox 2 amb el botó Borrar

A continuació s'ha implementat el mètode que s'executarà al fer clic sobre el botó que esborrarà tot el text, el codi és el següent:

```
private void Borrar_Click(object sender, EventArgs e)
{
    textBox2.Clear();
}
```

6.2.11. Implementació d'una petita rutina.

S'ha decidit crear un botó que al fer-hi clic el robot e-puck segueixi la següent rutina fins que es torni a pulsar el botó:

1. Girar a la dreta durant dos segons amb tots els leds encesos i reproduint el so 2.
2. Aturar-se dos segons amb tots els leds apagats i reproduint el so 3.
3. Girar a l'esquerre durant dos segons amb tots els leds encesos i reproduint el so 4.
4. Aturar-se dos segons amb tots els leds apagats i reproduint el so 5.
5. Tornar a començar

Per poder utilitzar aquesta rutina que s'executa sense fi s'ha utilitzat el mateix sistema que per cridar la rutina que mostra els paràmetres de l'acceleròmetre. S'ha creat un *Thread* fill que executa la rutina i es controlen les pulsacions del botó mitjançant un booleà.

La rutina consisteix en un bucle sense fi que crida els mètodes necessaris per que el robot realitzi la rutina que s'ha explicat avanç com es veu en el codi següent:

```
private void RutinaProva()
{
    while (true)
    {
        epuck.SetBodyLed(0);
        epuck.SetFrontLed(0);
        epuck.PlaySound(0);
        epuck.SetSpeed(0, 0);
        epuck.SetLed(8, 0);
        Thread.Sleep(2000);
        epuck.SetBodyLed(1);
        epuck.SetFrontLed(1);
        epuck.PlaySound(2);
        epuck.SetSpeed(500, -500);
        epuck.SetLed(8, 1);
        Thread.Sleep(2000);
        epuck.SetBodyLed(0);
        epuck.SetFrontLed(0);
        epuck.PlaySound(3);
        epuck.SetSpeed(0, 0);
        epuck.SetLed(8, 0);
        Thread.Sleep(2000);
        epuck.SetBodyLed(1);
        epuck.SetFrontLed(1);
        epuck.PlaySound(4);
        epuck.SetSpeed(-500, 500);
        epuck.SetLed(8, 1);
        Thread.Sleep(2000);
        epuck.SetBodyLed(0);
        epuck.SetFrontLed(0);
        epuck.PlaySound(5);
        epuck.SetSpeed(0, 0);
        epuck.SetLed(8, 0);
        Thread.Sleep(200);
    }
}
```


6.2.12. Disseny dels elements de connexió

Per poder connectar amb el robot s'ha decidit crear un botó de connexió i un *TextBox* que també es troba a la llista de l'esquerra de la visió de disseny tal com es veu a la figura 6.2.

En aquest *Textbox* l'usuari podrà escriure el port al qual es troba el robot e-puck. Aquest element té el nom *textBox1* i el text inicial *COM5* que és el port que s'ha decidit que es connecta l'epuck en tots els apartats del projecte. El botó té el nom *buttonConnect*, el text *Connect* i un tamany 70,20. També s'ha afegit un element de tipus *Label* que serveix per afegir text descriptiu, que en aquest cas es *Com Port*.

Un cop s'han afegit els elements necessaris s'han ordenat tal com es mostra a la figura 6.10:

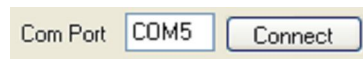


Figura 6.10: Distribució dels elements de connexió

6.2.13. Implementació del mètode de connexió

Per connectar amb el robot e-puck s'ha creat el mètode que s'executarà al fer clic sobre el botó de la mateixa manera que els anteriors. Aquest botó servirà tant per connectar com per disconnectar per tant el primer que es fa en aquest mètode és un condicional que distingeix entre si està connectat o no mitjançant el mètode *Connected()* de *EpuckComm*.

Si no està connectat cridarà el mètode *connectToEpuck* amb el port entrat a *textBox1*, també es canvia el text del botó a *Disconnect* i s'activen tots els botons de la interfície (veure figura 6.11) que estan desactivats fins que no es connecta amb el robot, per tal d'evitar que s'intentin enviar ordres avanç d'estar connectat ja que això provocaria errors (veure figura 6.12).

Si ja està connectat primerament es desactivaran tots els botons per evitar errors, tot seguit es mataran els subprocessos que estiguin activats, a continuació es cridarà el mètode *Disconnect()* de *EpuckComm* per realitzar la desconexió i es canvia el text del botó a *Connect*.

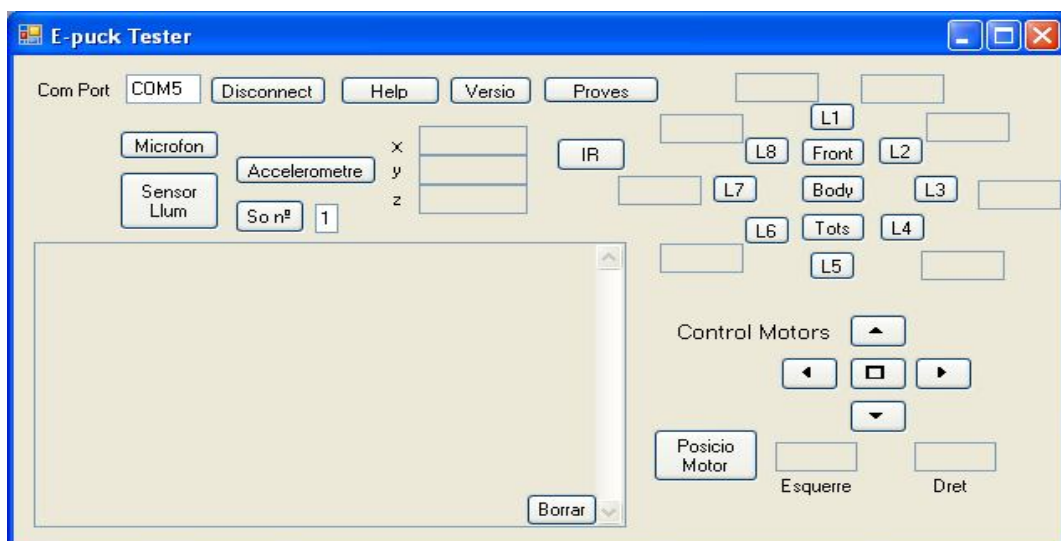


Figura 6.11: Distribució final de l'aplicació un cop s'ha connectat

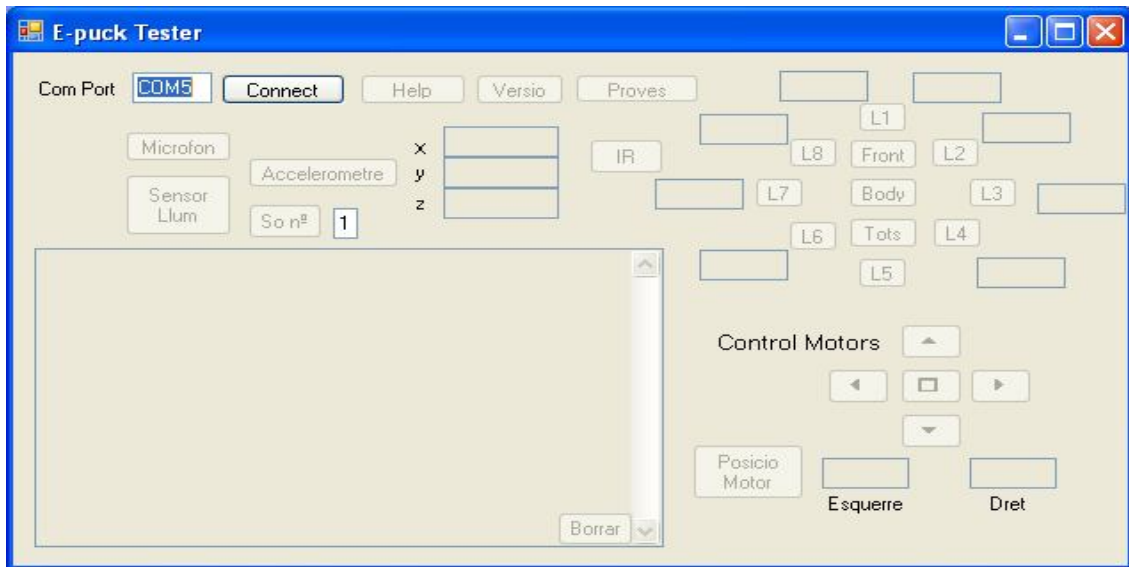


Figura 6.12: Distribució final de l'aplicació avanç de connectar

7. Resultats del programa de control automàtic sobre el robot real

7.1 Introducció

Per tal de poder comparar el funcionament del robot en la simulació amb el robot real es va decidir que s'executaria el mateix algorisme que s'ha executat en l'apartat 4 d'aquest projecte com es pot veure en el diagrama de la figura 7.1

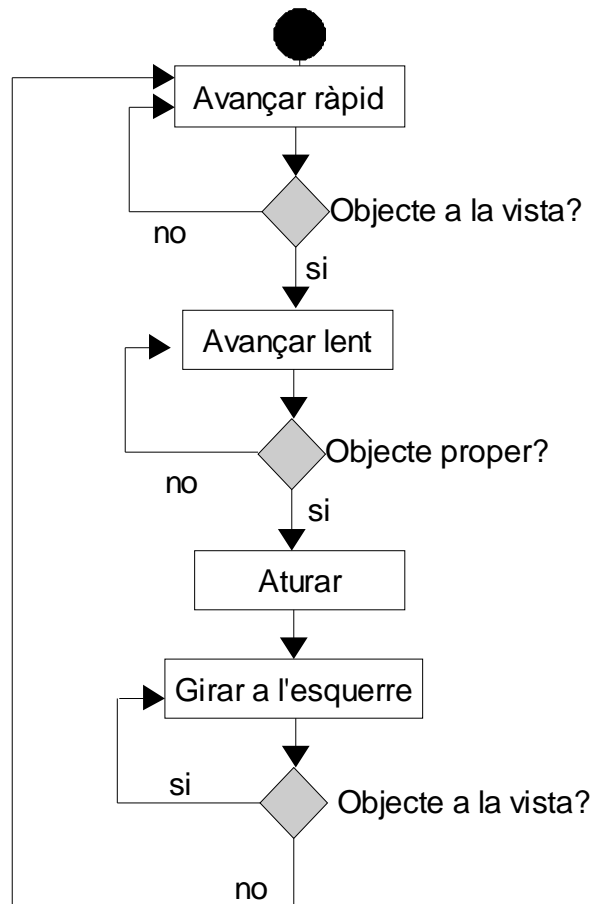


Figura 7.1: Diagrama de flux de l'algorisme que seguirà el robot e-puck real

7.2 Implementació

7.2.1. Crear el projecte i afegir els elements necessaris

El primer que es va fer va ser crear un nou projecte que s'ha anomenat *AutonomRealEpuck* utilitzant la plantilla *SimpleDssService*. A continuació s'ha afegit la classe *EpuckComm* al projecte tal com s'ha explicat anteriorment per tal de poder tenir accés al robot e-puck utilitzant el bluetooth. Per utilitzar-la primer s'ha creat una instància dins la declaració i s'ha afegit la següent línia de codi al mètode *Start()*:

```
epuck = new EpuckComm();
```

7.2.2 Implementació del mètode Start()

Per tal de controlar el robot e-puck real de manera que actuï com l'algoritme que s'ha explicat anteriorment s'ha modificat el mètode *Start()* implementant una màquina d'estats. Per fer això s'ha utilitzat un enter que s'ha anomenat *LogicalState* per indicar en quin estat es troba, a continuació s'ha creat una estructura de tipus *switch* amb quatre casos segons el valor del *LogicalState*.

En el primer cas es mira si l'e-puck es troba prop d'algun obstacle, si és així passarà al següent estat, si no posarà els motors a alta velocitat.

En el segon cas es mira si l'e-puck es troba molt proper de l'obstacle, si és així passarà al següent estat, si no posarà els motors a baixa velocitat.

En el tercer cas es posen els motos de l'e-puck a velocitat 0, és a dir, s'atura el robot, i es passa al següent estat.

En el quart cas es mira si l'e-puck es troba prop d'algun obstacle, si es així es posarà el motor dret a una velocitat mitjana i el motor esquerre a la mateixa velocitat en valor absolut però amb signe negatiu de manera que el robot girarà sobre si mateix cap a l'esquerre i si no hi ha cap obstacle es passarà al primer estat.

Aquest *switch* s'ha posat dintre d'un *while(true)* perquè s'executi constantment juntament amb una funció que actualitza la distància que s'explicara en el següent apartat.

El codi final en C# del mètode *Start()* és el següent:

```
protected override void Start()
{
    epuck = new EpuckComm();
    epuck.conectToEpuck(5);
    epuck.CalibrateProxSensors();
    int LogicalState = 1;
    int distància;
    while (true)
    {
        distància = IR0167();
        switch (LogicalState)
        {
            case 1:
                if (distància > disSegura)
                    LogicalState++;
                else
                    epuck.SetSpeed(500, 500);
                break;
            case 2:
                if (distància > disAturar)
                    LogicalState++;
                else
                    epuck.SetSpeed(100, 100);
                break;
            case 3:
                epuck.SetSpeed(0, 0);
                LogicalState++;
                break;
            case 4:
                if (distància < disSegura)
```

```

        LogicalState = 1;
    else
        epuck.SetSpeed(-100, 100);
        break;
    }
}
}

```

7.2.3. Implementació del mètode per obtenir la distància

Com en el cas de la simulació s'ha considerat com a distància la suma dels valors que retornen els sensors IR0, IR1, IR6 i IR7, és a dir:

$$\text{Distància} = (\text{IR0} + \text{IR1} + \text{IR6} + \text{IR7})$$

Per obtenir aquest valor s'ha creat un mètode que s'ha anomenat *IR0167()*. Aquest mètode fa un recorregut del *string* que retorna el mètode *GetProximitySensors()* de la classe *EpuckComm*, en aquest recorregut es busquen els valors que retornen els sensors IR desitjats, es sumen en un enter i es retorna el valor. El codi que realitza això es el següent:

```

private int IR0167()
{
    string s = epuck.GetProximitySensors();
    int y=0;
    if (s.Length > 12 && s[0] == 'n' && s[1] == ',')
    {
        char c;
        int j = 1;
        string a;
        int x;
        for (int i = 0; i <= 7; i++)
        {
            x = 0;
            j++;
            c = s[j];
            a = "";
            while (c != ',' && c != 'r' && c != '\n')
            {
                a += c;
                j++;
                c = s[j];
            }
            if (i == 0 || i == 1 || i == 6 || i == 7)
            {
                int.TryParse(a, out x);
                y = y + x;
            }
        }
    }
    return y;
}

```

7.2.4. Elecció de les distàncies de seguretat i d'aturada.

Per tal de poder escollir les distàncies d'aturada i de seguretat primerament s'ha mirat com funcionaven els sensors IR i quins valors retornaven.

Mirant la gràfica i la taula del apartat 2.3.2 sobre sensors infrarojos (veure figura 7.2) s'ha decidit que la distància de seguretat podria ser d'uns tres centímetres per tant el valor que s'ha posat com a distància de seguretat és 300, que és el valor que s'obté quan el robot està a una distància de tres centímetres.

De la mateixa manera s'ha decidit que el màxim que es pot acostar el robot al obstacle és de dos centímetres, per tant el valor de la velocitat d'aturada ha de ser propera a 500.

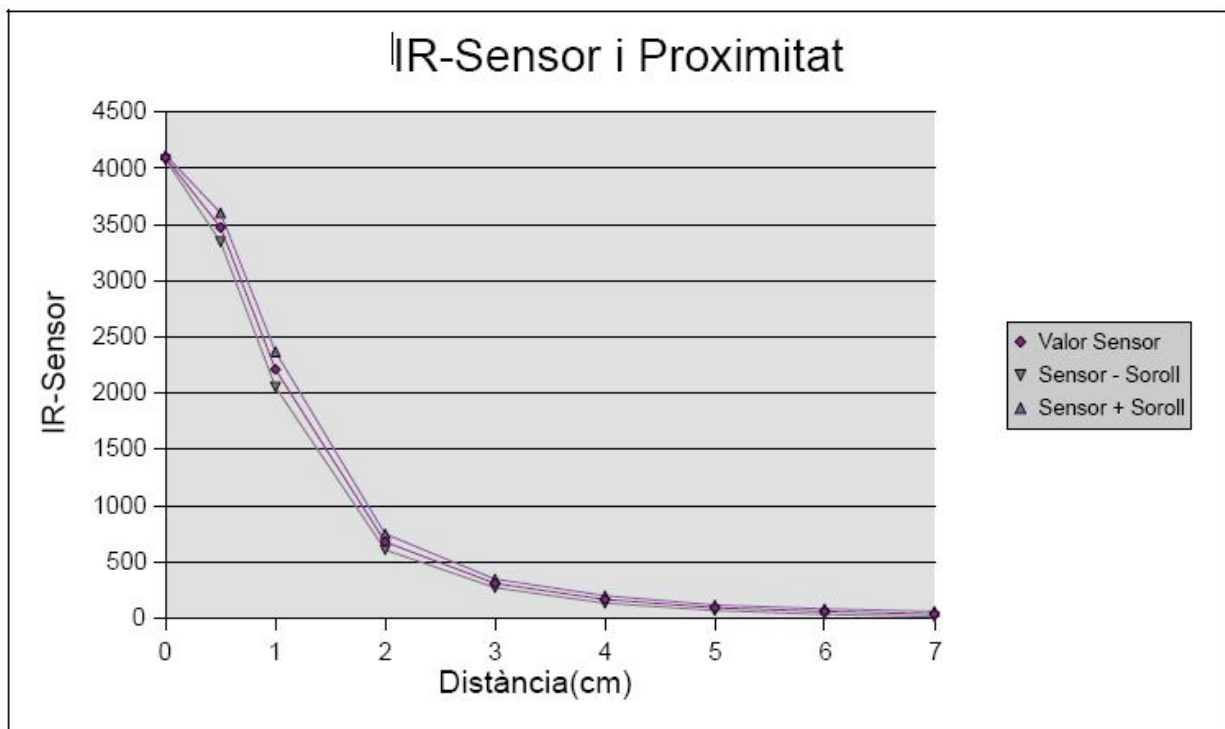


Figura 7.2: Gràfica de relació dels valors obtinguts per un sensor IR i la Distància que es troba l'objecte

7.3 Resultats i conclusions

Després de provar el programa sobre el robot real s'ha vist que el robot no sempre s'atura avanç de topar a l'obstacle però la majoria de vegades ho fa sense problemes, això també passa alguns cops en la simulació (figura 7.3).

També s'ha vist que en algunes ocasions es posa en l'estat dos massa aviat i va a poc a poc fins a trobar un obstacle, aturar-se, girar i tornar a la velocitat alta. Això no passava en la simulació ja que el sensor d'infrarojos *sickLRF.Y2005* és molt més potent i fiable, ja que pot veure objectes en un rang de distàncies més ampli i era més fàcil diferenciar entre tenir el camp lliure i tenir un obstacle a prop.

A part d'aquests errors que no sempre succeeixen, la majoria de vegades funciona igual que en la simulació.

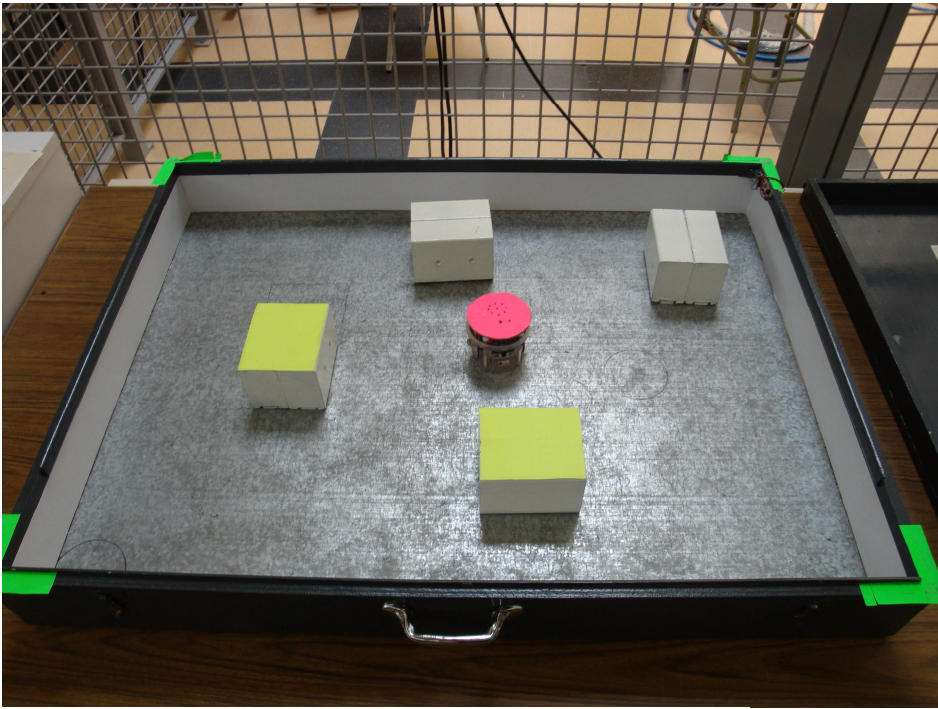


Figura 7.3: E-puck executant el programa de control

8. Conclusions i treballs futurs

8.1 Valoració global

Primerament el que s'ha après en la realització d'aquest projecte és el funcionament d'un robot mòbil, ja que fins a la realització del projecte només s'havia treballat amb braços robòtics. S'ha après el robot e-puck, el funcionament de tots els seus sensors i actuadors, s'ha estudiat el seu codi intern i el protocol de comunicació que utilitza. I finalment s'ha vist com treballar sobre el robot real veient els errors que es poden trobar i com solucionar-los.

En el transcurs del projecte també s'ha vist gran part del potencial de l'eina de programació de robots *Microsoft Robotics Studio*.

S'ha vist la potencia de l'entorn de simulació, la possibilitat de simular robots, entorns, objectes, etc. de forma molt realista i fidel a la realitat. També s'ha pogut observar la fidelitat en que les forces són representades: gravetat, fregaments, rebots, etc. de manera que la simulació és molt similar a la realitat a diferència d'altres entorns de simulació que no cuiden tant aquests detalls, gràcies a això es podria arribar a substituir completament els robots reals per les simulacions.

Tot i que no s'ha utilitzat per la realització dels temes del projecte, durant el transcurs del estudi dels elements de MRS s'ha utilitzat l'entorn de programació visual i s'han après alguns dels conceptes d'aquest.

També s'ha après a treballar amb un llenguatge nou, el C#, orientat a serveis que no s'havia après durant la carrera, utilitzant l'entorn de programació *Microsoft Visual C#* on s'han vist totes les facilitats que ofereix aquest entorn, llibreries, plantilles, disseny d'aplicacions Windows, manipulació de fitxers xml, la compilació de projectes i l'execució d'aquests, etc. Tot implementant els programes s'han vist tots els elements d'aquest llenguatge i que les parts més bàsiques tenen gran similitud amb el llenguatge C i s'han vist les diferències entre els dos. Una part molt important que s'ha après durant la implementació és la detecció, comprensió i correcció d'errors, ja que se n'han comès molts, alguns durant la compilació i d'altres durant l'execució.

Finalment i no menys important s'ha après a escriure una memòria d'un projecte una mica llarg, que no resulta una tasca senzilla per algú que no és gens de lletres. Però s'ha vist la importància de documentar els treballs realitzats per poder facilitar la comprensió d'aquests.

8.2. Assoliment d'objectius

Primer objectiu: Estudi de Microsoft Robotics Studio: aprendre a utilitzar els seus components i entendre el funcionament intern.

Per assolir aquest objectiu s'ha estudiat tota la documentació que està disponible a la pagina web i en els documents d'ajuda del mateix programa, s'ha llegit un llibre de programació de MRS i s'han realitzat tots els tutorials disponibles, de programació visual, de serveis, de hosting, robòtics i de simulació. Els robòtics no es van posar a prova ja que requerien robots reals que no són els que es tenen al laboratori de robòtica. Un cop llegida la documentació i després de realitzar els tutorials es va assolir molt bé aquest objectiu, tot i que durant la realització del projecte s'han consultat novament alguns conceptes i s'han llegit fòrums, alguns de la pàgina oficial i d'altres de pàgines externes.

Segon objectiu: Modelar i simular el robot e-puck amb els seus actuadors (2 motors) i sensors bàsics (sensors d'infrarojos).

S'ha creat una simulació força fidel a la realitat, s'ha pogut crear un robot amb dos rodes amb motors i se li ha donat una aparença força fidel a la del robot real. Se li ha assignat un pes i unes mesures realistes i un fregament bastant real, també s'ha buscat i assignat el centre de gravetat.

En el cas dels sensors infrarojos, s'ha buscat una solució alternativa ja que MRS no disposa d'aquest tipus de sensors, finalment es va decidir utilitzar un sensor de tipus Laser Range Finder, un sensor que dona les distàncies a objectes en un rang de 180°, i es va decidir situar-ne dos al centre, un per els sensors frontals i un altre per els sensors de la part posterior i d'aquests només utilitzar-ne el valor dels angles que corresponen als sensors del robot. La diferència final és que aquest sensors són molt més fiables, precisos i de major abast que els reals.

També, tot i no forma part de l'objectiu inicial, s'hi ha afegit una càmera web com la del robot. I s'ha creat un entorn similar al que s'ha creat en el laboratori de robòtica.

Tercer objectiu: Realitzar un programa que controli el robot de forma autònoma en el simulador i comprovar-ne el seu funcionament. Aquest programa ha de moure el robot sense topar amb els objectes que trobi.

Per assolir aquest objectiu primerament s'ha dissenyat un programa de tipus màquina d'estats, després s'ha estudiat com comunicar amb els serveis de la simulació de manera que es puguin donar ordres i obtenir informació de l'entorn, en el nostre cas les distàncies que llegeix el làser. Un cop vist això s'ha implementat i s'han realitzat les proves. Finalment s'han estudiat el resultat i s'ha vist que en la majoria de vegades s'assoleix l'objectiu però en algunes ocasions el robot topa amb les parets i s'encalla.

Quart objectiu: Dissenyar i implementar la interfície de comunicació entre el robot real i MRS.

Per comunicar-se amb el robot es va crear una classe que implementés els mètodes per aconseguir la informació dels sensors i donar ordres als actuadors. Un cop implementada la classe l'objectiu es va complir completament.

Cinquè objectiu: Realitzar proves amb el robot real, analitzar i valorar el funcionament de MRS.

La realització del programa no ha portat gaires problemes ja que un cop s'ha realitzat bé el disseny de l'algorisme, la implementació ha resultat fàcil, ja que es disposava dels mètodes necessaris per realitzar-lo amb facilitat.

Un cop implementat les proves han estat satisfactòries, no s'han obtingut errors greus i s'han resolt fàcilment els problemes.

Objectiu extra: Realitzar una aplicació per llegir tots els sensors del robot e-puck i donar ordres de modificació dels actuadors.

Durant la realització del projecte es va decidir crear una aplicació que mitjançant botons fos possible obtenir els valors que obtenen els sensors del robot e-puck, moure el robot, encendre els leds i reproduir sons en l'altaveu.

Es va decidir que alguns dels sensors donessin la informació contínuament i aquí és l'únic lloc on sorgeixen errors si es premen dos botons d'aquests que s'executen contínuament alhora. Tota la resta funciona perfectament.

Finalment podríem dir que s'han assolit tots els objectius que s'havien plantejat al inici del projecte.

8.3. Treballs futurs

Seguint la mateixa linea del projecte es poden fer algunes millores en tots els àmbits ja que únicament s'han treballat totes les parts d'una manera simple i poc profunda.

Es podrien crear els sensors que falten en la simulació per tal de poder-les afegir al robot, crear una simulació completa i poder realitzar programes que utilitzin tots els sensors. Una altra part de la simulació que es podria realitzar és la creació dels leds i la seva il·luminació.

Es pot millorar la imatge del robot en la simulació perquè sigui molt més realista.

Es podrien realitzar programes d'una alta complexitat sobre la simulació i/o sobre el robot real utilitzant com a base el que s'ha creat en aquest projecte, ja que els programes que s'executen sobre el robot són molt senzills.

Es podrien realitzar programes sobre el robot real que utilitzin altres sensors i actuadors que els que s'han utilitzat en aquest projecte.

Es podrien realitzar programes que controlin multitud de robots alhora ja sigui a la simulació o en la realitat de manera que interaccionin entre ells, ja que es disposa de més robots e-puck en el laboratori.

9. Bibliografia

Llibre consultat:

Sara Morgan. *Programming Microsoft Robotics Studio 2008*. Microsoft Press, 2008.

Publicació consultada:

Jackson, J.; **Microsoft robotics studio: A technical introduction**, Robotics & Automation Magazine, IEEE Volume 14, Issue 4, Dec. 2007 Pàgines: 82 - 87
Digital Object Identifier 10.1109/M-RA.2007.905745

PFC consultat:

Odometria i planificació de trajectòries amb el robot e-puck. Arnau Carrera Viñas.
Desenvolupat l'any 2009 en la titulació Eng. Tècn. Informàtica de Sistemes.

Webs consultades:

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

<http://msdn.microsoft.com/en-us/vcsharp/default.aspx>

<http://www.benaxelrod.com/MSRS/>

<http://www.e-puck.org/>

<http://www.cyberbotics.com/>

ANNEX A. MANUAL D'USUARI

A.1. Instal·lació Microsoft Robotics Studio

Durant tot el projecte, tant a la recerca com a la elaboració dels programes s'ha utilitzat la versió **1.5 Refresh** de Microsoft Robotics Studio. En el moment d'escriure això existeix una *Community Technical Preview* de la versió 2008 del programa, és una versió més nova però no és final. Per aquesta raó i per la popularitat de la versió 1.5 (els fitxers entre una i altre versió no sempre són compatibles) s'ha decidit fer servir aquesta. Primer de tot s'ha d'instal·lar Microsoft Robotics Studio.

La versió **Microsoft Robotics Studio 1.5 Refresh** es pot descarregar directament des de la següent direcció, a la plana web de Microsoft:

<http://www.microsoft.com/downloads/details.aspx?familyid=73092ff6-e37b-45c6-8e5e-c23d5d632b1e&displaylang=en&tm>

És un instal·lador de 87Mb instal·lable a ordinadors amb sistema operatiu: Windows CE; Windows Server 2003 R2 (32-Bit x86); Windows Server 2003 R2 x64 editions; Windows Vista; Windows XP; Windows XP 64-bit. (Figura A.1)

Aquest instal·lador i tots els que necessaris també es troba dintre de la carpeta \instalacio.

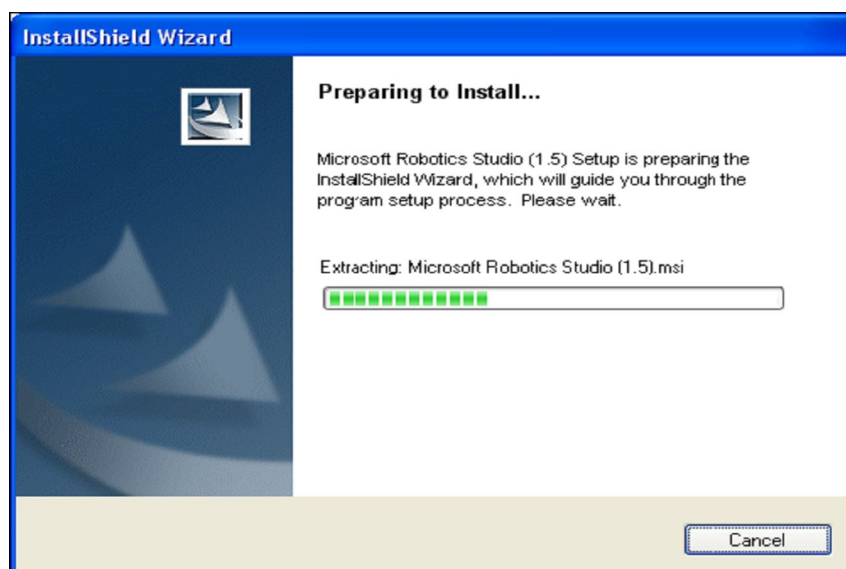


Figura A.1. Instal·lació Microsoft Robotics Studio

Un cop s'inicia el procés l'aplicació informa a l'usuari de tot el software addicional necessari i dona la possibilitat d'instal·lar-ho, com són: la plataforma .NET 3.0, Microsoft XNA 2.0 o Ageia Physics Engine. Per defecte el programa s'instal·la a la unitat C a la ruta: *C:\Microsoft Robotics Studio (1.5)*

En segon lloc, per a la versió 1.5 Refresh hi ha dos paquets d'actualització, amb els quals instal·lats s'ha fet tot el treball, així que recomano la seva instal·lació un cop instal·lat tot l'anterior. Els paquets són:

Runtime and Tools Update for Microsoft Robotics Studio (1.5):

<http://www.microsoft.com/downloads/details.aspx?FamilyId=2F747403-7D94-43F0-836B-84247FEE66C6&displaylang=en>

Samples Update for Microsoft Robotics Studio (1.5):

<http://www.microsoft.com/downloads/details.aspx?FamilyId=7EEB9B70-0E86-4E3E-92AF-6148BDA34B7C&displaylang=en>

Si després d'haver realitzat tots els passos sorgeix algun problema, o durant l'execució de l'aplicació, Microsoft disposa d'un portal anomenat MSDN (*Microsoft Developer Network*), el qual inclou fòrums en els que es pot trobar ajuda de tota mena, tant a nivell pràctic com tècnic. Jo mateix m'he trobat amb alguns problemes que m'han solucionat als fòrums. En aquesta direcció es poden realitzar cerques al fòrum:

<http://forums.microsoft.com/qawizard/ask.aspx?siteid=1>

Un cop instal·lada l'aplicació es podrà accedir als seus elements des de una nova carpeta a: *Inicio / Programes / Microsoft Robotics Studio (1.5)*.

Veure elements instal·lats a la figura A.2.

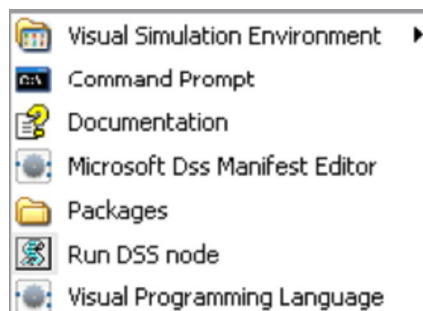


Figura A.2. Elements de *Microsoft Robotics Studio*

A.2. Instal·lació de Microsoft Visual C#

La part de programació del projecte s'ha realitzat utilitzant la versió 2005 Express Edition d'aquest entorn de programació. Es pot trobar en aquesta direcció:

<http://www.microsoft.com/express/download/>

O també a la carpeta `\instalacio` amb el nom `vcsetup`.

Un cop descarregat s'han de seguir els passos per instal·lar la plataforma.

També es necessari instal·lar el Microsoft® Visual Studio® 2005 Express Editions Service Pack 1 que es pot trobar a la següent pagina.

<http://www.microsoft.com/downloads/details.aspx?FamilyId=7B0B0339-613A-46E6-AB4D-080D4D4A8C4E&displaylang=en>

O a la carpeta `\instalacio` amb el nom `vs80sp1-kb926749-x86-intl`

Després d'instal·lar-la es necessari tornar a executar el fitxer d'instal·lació de MRS per tal de que s'instal·lin les llibreries i plantilles al Visual C# necessàries per la programació de serveis DSS.

A.3. Còpia dels projectes

El següent pas és copiar les carpetes que es troben a la carpeta `/instalacio/projectes` i posar-les a la carpeta `Microsoft Robotics Studio (1.5)/projects`.

També cal posar el fitxer `miniepuck.bos` a la carpeta `Microsoft Robotics Studio (1.5)/store/media`

A.4. Configurar Bluetooth

Per poder executar els programes del projecte es necessari configurar el bluetooth per que realitzi la connexió amb el robot e-puck utilitzant el port COM5. Per fer això cal seguir els següent passos:

1. Encendre el robot d'e-puck. La llum verda hauria d'encendre's.
2. Al ordinador, encendre organitzador Bluetooth
3. Cal esculli l'aparell d'e-puck_XXXX on és XXXX el número del e-puck (veure figura A.3). El número és escrit a la placa metàl·lica sota l'altaveu.(Figura A.4)



Figura A.3 : Elecció de robot e-puck

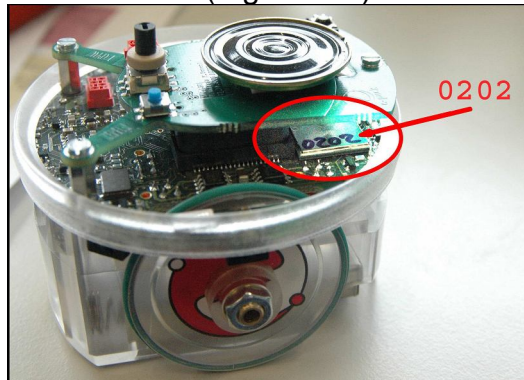


Figura A.4: Situació del número del robot

4. En el següent quadre cal introduir el codi de anterior per accedir al e-puck. Figura A.5.

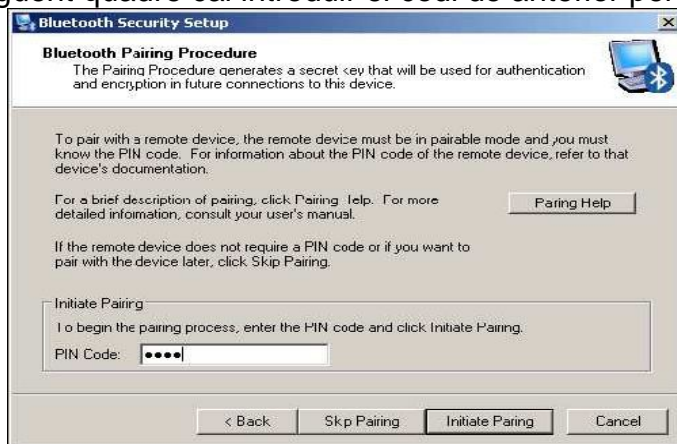


Figura A.5: Introducció del codi PIN

6. Seleccionar el servei de COM5 del robot d'e-puck. Figures A.6 i A.7.

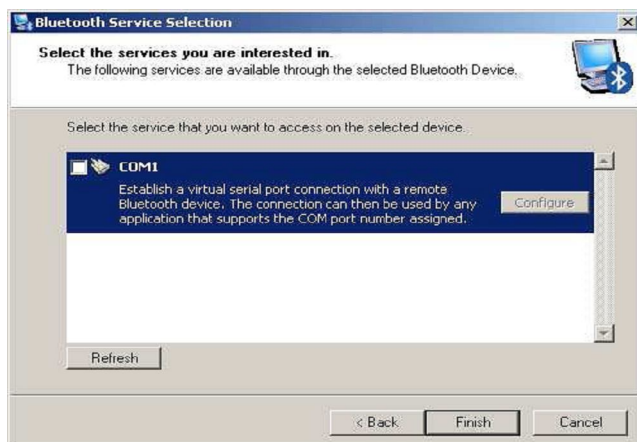


Figura A.6: Seleccionar Port

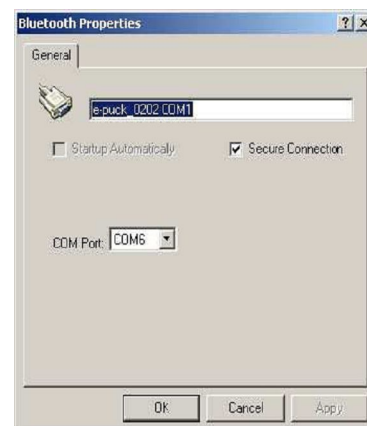


Figura A.7: Propietats del Port

A.5. Execució dels programes.

Per executar els programes hi ha dues opcions, la primera vegada es necessari fer-ho de la primera manera i abans d'executar el projecte *AutonomEpuck* per primera vegada es necessari que primer s'hagi executat el projecte *simEpuck* ja que el primer utilitza el segon i d'aquesta manera s'haurà compilat i creat el servei. Un cop executats de la primera forma ja es podrà fer sempre de la segona.

a)Utilitzant la plataforma de programació Visual C#

Fen servir aquesta opció es pot veure el codi, les classes, etc. del programa. Primer cal obrir la plataforma, a continuació fer clic a *Archivo>Abrir Proyecto*. Anar a *C:\Microsoft Robotics Studio (1.5)\projectes* obrir la carpeta del projecte que es vulgui executar i obrir el fitxer de projecte, com es mostra a la figura A.8.

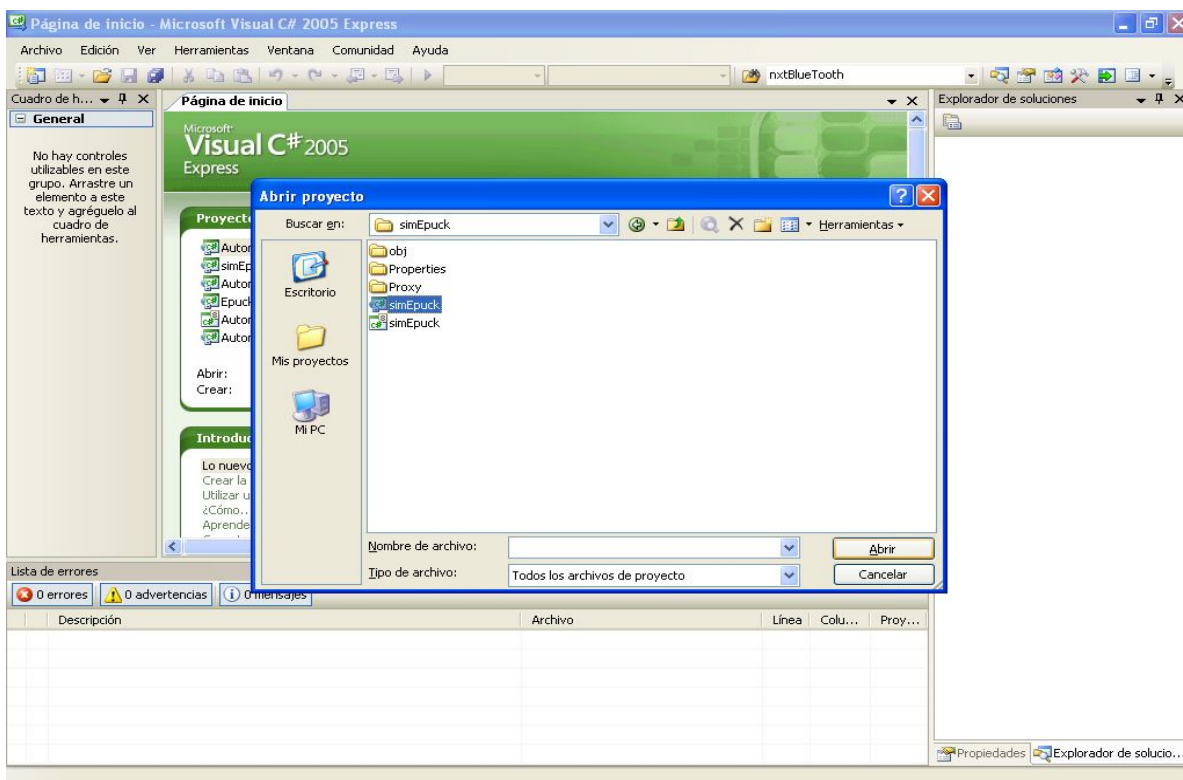



Figura A.8: Obrir Projecte

Un cop obert el projecte l'únic que cal fer per executar-lo és fer clic sobre el boto  o fer F5, el Visual C# compilarà i executarà el projecte. Ames de crear el fitxer Dss.

b) Utilitzant el Microsoft Dss Manifest Editor

Primer cal obrir el Microsoft Dss Manifest Editor. Figura A.9

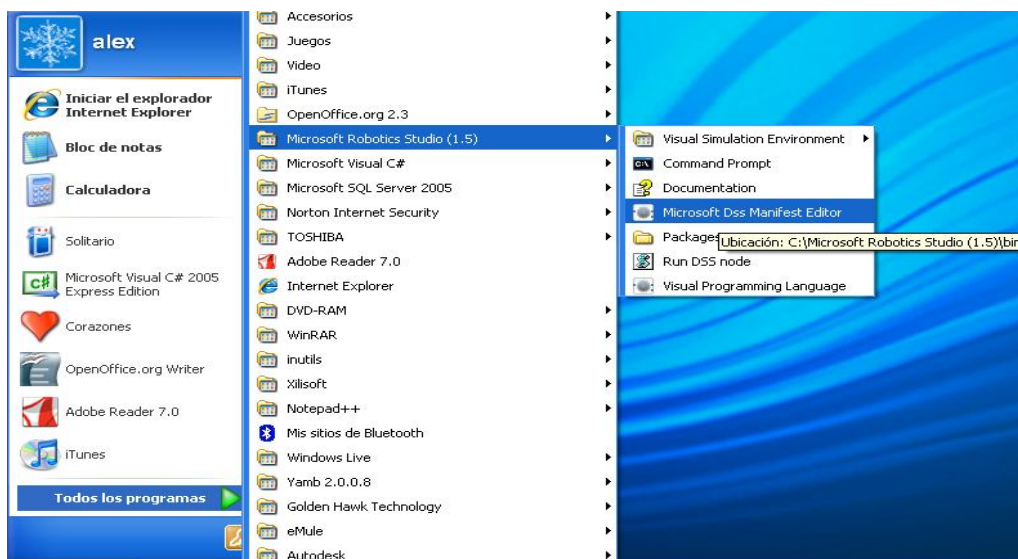


Figura A.9: Obrir Microsoft Dss Manifest Editor

Un cop obert el cal fer es buscar de la llista de l'esquerre el servei que es vol executar i arrossegar-lo al quadre central *Manifest* a continuació cal fer *Run>Run Manifest* o prémer F5 per començar a executar. Figura A.10.

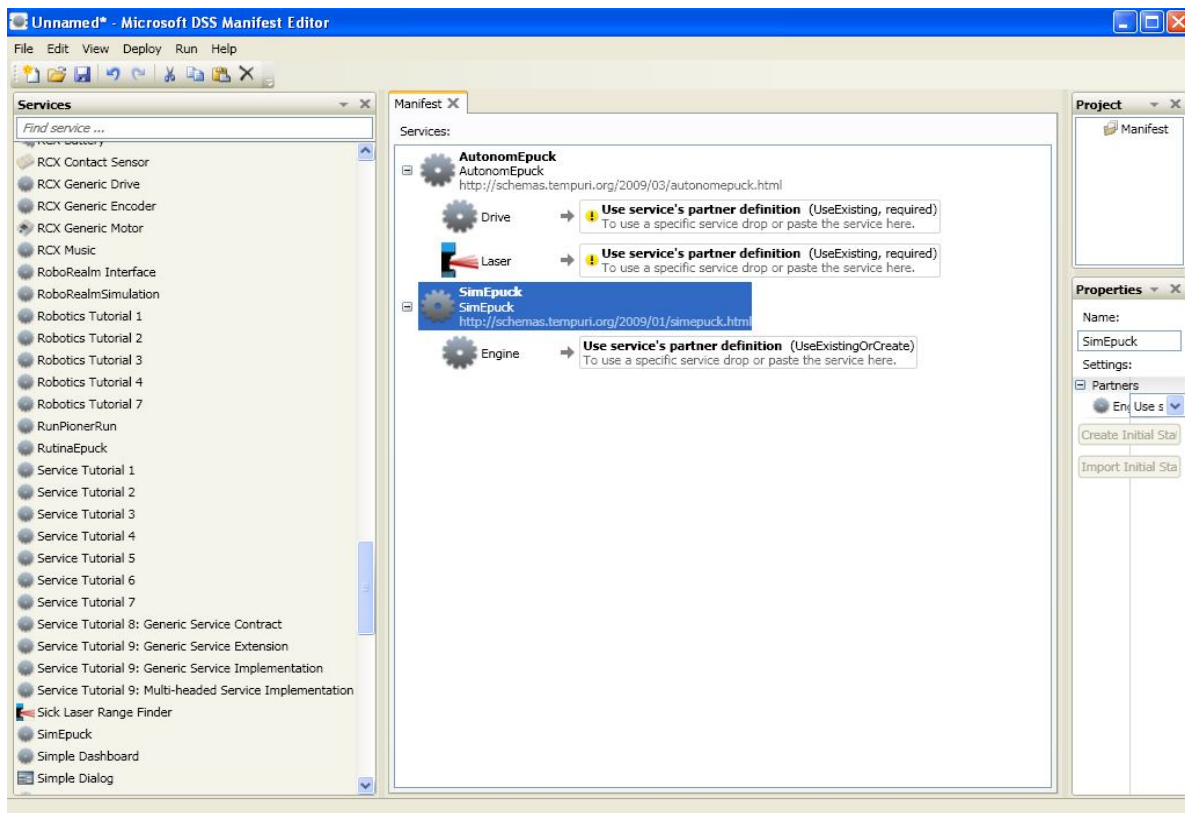


Figura A.10: Interfície Microsoft Dss Manifest Editor

Nota: En el cas de voler executar *AutonomEpuck* es necessari afegir al manifest *SimEpuck*. I el projecte *EpuckBluetooth* no es pot executar d'aquesta manera.