View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Universidad Carlos III de Madrid e-Archivo

Universidad Carlos III de Madrid
Proyecto Fin de Carrera
Departamento de Informática

Tutor: Raúl Arrabales Moreno
Directora: Clara Benac Earle

Utilización de la herramienta Microsoft Robotics Developer Studio en la competición Imagine Cup '09

Alumno: Álvaro Fernández Díaz
email: 100055116@alumnos.uc3m.es

Agradecimientos

A mis padres, Marcelino y Begoña, por todo lo que se han esforzado porque llegue este momento. Este proyecto es, en buena medida, fruto de su perseverancia y trabajo.

A mi hermana, Sara, por comprenderme cuando el estrés de las tareas se imponía a la razón.

A mis tíos, Jaime y Ana, por el apoyo que me aportaron, y siguen aportando, desde el principio de mi carrera.

A mis familiares, amigos y todos aquellos que, con sus palabras de ánimo, me infundieron la fuerza necesaria para conseguir mis objetivos.

A mis tutores, Clara y Raúl, por el apoyo aportado durante el desarrollo vertiginoso del proyecto.

Índice

Capítulo 1: Introducción ... 12

1.1.- Análisis y comprensión de la tecnología MRDS 2008 ... 13

1.2.- Participación en la competición Imagine Cup 2009 ... 14

Capítulo 2: Estado de la cuestión Navegación Robótica ... 16

2.1.- Mapeo ... 17

2.1.1 Representaciones continuas ... 19

2.1.2.- Estrategias de descomposición ... 21

2.1.3.- Retos actuales y nuevas líneas de investigación ... 25

2.2.- Localización ... 25

2.2.1.- Localización de Markov ... 28

2.2.2 Filtro de Kalman ... 31

2.3.- Localización y Mapeo Simultáneo: SLAM ... 31

2.4.- Planificación ... 33

2.4.1.- Grafo de Visibilidad ... 34

2.4.2.- Diagrama de Voronoi ... 34

2.4.3 Método del gradiente ... 35

2.5.- El algoritmo Wedgebug ... 36

Capítulo 3: Herramientas ... 40

3.1.-Microsoft Visual Studio 2008 ... 40

3.2.-Microsoft Robotics Developer Studio 2008 ... 40

3.2.1.- Decentralized Software Services ... 41

3.2.2.- Concurrency and Coordination Runtime ... 43

3.2.2.1.- Puertos ... 44

3.2.2.2.- Árbitros y Receptores ... 44

3.2.2.3.- Tareas, colas de ejecución y dispensadores ... 45

Capítulo 4: La competición Imagine Cup 48

4.1.- Historia y objetivos 48

4.2.- Planteamiento del problema 49

4.2.1.- Descripción general 49

4.2.2.- Robot NASA Mars Rover 50

4.2.2.1.- Tracción 50

4.2.2.2.- Cabeza articulada 51

4.2.2.4.- Cámaras 52

4.2.2.5.- Espectrómetro 53

4.2.2.6.- Puerto de comunicaciones 54

4.3.- Sistema de puntuación 55

4.4.- Contexto real 56

4.4.1.- La misión Mars Exploration Rover 56

4.4.2.- Exploración del cráter Endurance y el escudo térmico 59

Capítulo 5: Código de Partida 62

5.1.- Servicio ImageProcessing 64

5.1.1.- Fichero ImageProcessingTypes 65

5.1.2.- Fichero ImageProcessing.cs 68

5.1.2.1.- Estado del servicio 68

5.1.2.2.- Puerto Principal 69

5.1.2.3.- Servicios Asociados 69

5.1.2.4.- Manejadores 70

5.1.2.5.- Lógica de ejecución 71

5.2.- Servicio MarsChallenger 73

5.2.1.- Fichero MarsChallengeTypes.cs 74

5.2.2.- Fichero MarsChallenger.cs 76

5.2.2.1.- Estado del servicio 76

5.2.2.2 Puerto Principal 77

5.2.2.3.- Servicios asociados 77

5.2.2.4.- Manejadores 78

5.2.2.5.- Lógica de ejecución....................................... 80

5.2.2.6.- Funciones Auxiliares.. 80

Capítulo 6: Solución Implementada 82

6.1.- Navegación Global.. 82

6.1.1.- Comportamiento Navigate 84

6.1.2.- Comportamiento RockFinder 91

6.2.- Navegación Local: Mecanismos de soporte a la navegación........ 94

6.2.1.- Mecanismos de prevención de atasco....................... 94

6.2.2.- Mecanismos de prevención de vuelco 95

6.2.3- Sistemas de corrección de ruta 96

6.3.- Localización del Escudo Térmico.. 96

6.3.1.- Patrulla del área 96

6.3.2.- Detección y aproximación al escudo.......................... 97

6.4.- Condiciones de finalización de la misión 98

6.5.- Diagrama de estados: transición entre comportamientos. 98

6.6- Resultados 99

Capítulo 7: Conclusiones y Líneas Futuras.............................. 102

Capítulo 8: Bibliografía y Referencias 106

Apéndice A: Instalación del entorno 108

Apéndice B: Contenido del CD 112

Apéndice C: Planificación 114

Apéndice D: Código Implementado 116

Índice de Figuras

Figura 1.- Representación bidimensional de un escenario de interior 17

Figura 2.- Representación bidimensional simplificada de un entorno. 19

Figura 3.- Representación en líneas infinitas (b) del entorno (a) 20

Figura 4.- Descomposición exacta en celdas de un escenario 21

Figura 5.- Descomposición original en celdas 22

Figura 6.- Descomposición fija 22

Figura 7.- Descomposición adaptativa 23

Figura 8.- Rejilla de ocupación 23

Figura 9.- Mapa topológico 24

Figura 10.- Localización en mapa de celdas 26

Figura 11.- Localización en rejilla de ocupación 26

Figura 12.- Camino recorrido 27

Figura 13.- Nube de puntos en posiciones 2, 3 y 4 27

Figura 14.- Proceso de localización de Markov 30

Figura 15.- Mapa del entorno obtenido mediante una técnica SLAM 33

Figura 16.- Grafo de visibilidad 34

Figura 17.- Diagrama de Voronoi 35

Figura 18.- Método del gradiente 36

Figura 19.- Descripción del área en forma de cuña analizado 36

Figura 20.- Extracción de nodos algoritmo Wedgebug 37

Figura 21.- Proceso de adquisición de nodos mediante diversos escaneos 38

Figura 22.- Servicio DSS 42

Figura 23.- Coordinación multitarea en CCR 46

Figura 24.- Rueda del robot 50

Figura 25.- Mástil y cabeza del robot 51

Figura 26.- Brazo mecánico del robot 52

Figura 27.- Cámara del robot 53

Figura 28.- Espectrómetro del robot 54

Figura 29.- Puerto de comunicaciones del robot 55

Figura 30.- Fotografía del Meridiani Planum 57

Figura 31.- Representación de un robot Mars Rover 58

Figura 32.- Panorámica del cráter Endurance realizada por el robot Opportunity 59

Figura 33.- Roca del Escudo Térmico 60

Figura 34.- Escudo térmico en el que viajó el robot Opportunity 60
Figura 35.- Orquestación de los servicios 63
Figura 36.- Servicio ImageProcessing 64
Figura 37.- Servicio MarsChallenger 74
Figura 38.- Coordenadas Yaw, Pitch y Roll 83
Figura 39.- Sistema angular de valores de las coordenadas Yaw, Pitch y Roll, respectivamente 83
Figura 40.- Camino generado 86
Figura 41.- Representación sobre el escenario del camino elaborado por el robot 87
Figura 42.- Camino elaborado 88
Figura 43.- Sentido de giro del robot 89
Figura 44.- División en áreas de los cuadrantes 89
Figura 45.- División cuadricular de una imagen 92
Figura 46.- División en zonas de un fotograma 93
Figura 47.- Proceso de análisis de una muestra 93
Figura 48.- Recorrido de la patrulla 96
Figura 49.- Representación sobre el mapa del recorrido de patrulla 97
Figura 50.- Diagrama de transiciones entre comportamientos 99
Figura 51.- Clasificación de la segunda ronda 100
Figura 52.- Pantalla de descarga 109
Figura 53.- Ventana de instalación 109
Figura 54.- Carpeta "RoboChamps\Mars\Sample" 110
Figura 55.- Interfaz del proyecto 110
Figura 56.- Asignacion temporal de tareas 115
Figura 57.- Seguimiento de tareas 115

Glosario

Aplicación Web Híbrida: también conocida como MashUp, es un sitio o aplicación web que usa contenido de otras aplicaciones Web para crear un nuevo contenido completo, consumiendo servicios directamente, siempre a través de protocolo http.

Escudo térmico: capa protectora de una nave espacial cuya finalidad es proteger a ésta de las altas temperaturas, originadas por fricción con la atmósfera, a las que se ve sometidas al efectuar la entrada a un planeta.

GPS: del inglés Global Positioning System o Sistema de Posicionamiento Global. El cometido de estos dispositivos es proporcionar la posición de un determinado elemento en el espacio.

Grado de libertad: magnitud que pueden variarse independientemente. En robótica suele coincidir el número de grados de libertad con el número de articulaciones móviles.

Odometría: estudio de la estimación de la posición de vehículos con ruedas durante la navegación.

Sol: día marciano. Equivale a 24 horas 39 minutos 35,244 segundos, un 3% más largo que un día solar terrestre.

Capítulo 1: Introducción

En la actualidad la robótica es una ciencia aplicada a multitud de áreas tan variadas como la medicina, seguridad, entretenimiento, investigación, salvamento, etcétera. A pesar de ello, la programación robótica no siempre está al alcance de cualquiera, pues se precisan dispositivos físicos en los que comprobar si el comportamiento del robot se corresponde con el que se deseaba implementar. Hasta ahora, esta necesidad de dispositivos físicos se ha subsanado mediante la simulación de entornos virtuales, con los que se pretendía emular el comportamiento del robot en un entorno real. No obstante, la gran mayoría de estos entornos no está a disposición de cualquier persona que desee utilizarlos, sino que requieren su compra.

Además, es frecuente que, tras implementar un determinado comportamiento y comprobar su resultado en un entorno simulado, no sea posible trasladar el código compilado directamente a un dispositivo físico, sino que se requiera su nueva compilación o, incluso, la modificación del código para que esto sea posible. Debido a ello, es posible que existan discrepancias entre ambas versiones de compilación de un supuesto mismo comportamiento. La aplicación Microsoft Robotics Developer Studio (MRDS) intenta subsanar todas estas barreras proporcionando una versión gratuita para estudiantes, profesores e investigadores y permitiendo la utilización de un mismo código para el control de un determinado dispositivo tanto en un entorno real como simulado.

Actualmente, se realizan multitud de competiciones estudiantiles cuyo objetivo es fomentar la innovación y la pasión por la tecnología en las nuevas generaciones. Entre estas competiciones se encuentra la conocida como Imagine Cup, organizada por Microsoft. Ésta es, posiblemente, la más extendida mundialmente debido a la gran inversión realizada en ella por parte de diversos colectivos y empresas, como Microsoft o Paramount Digital Entertainment, y a la amplia gama de secciones y modalidades que engloba. Para participar en este tipo de competiciones, se precisa poseer interés en algún aspecto de la tecnología que se encuentre reflejado en ellas, como es el caso de la programación robótica. Sin embargo, se requiere poseer ciertos conocimientos previos que permitan llevar a cabo las tareas necesarias para participar en dichas competiciones. Por tanto, resulta frecuente la realización de procesos de documentación e investigación con la finalidad de adquirir los mencionados conocimientos y habilidades.

En el presente proyecto, se muestra el trabajo realizado por el autor en la competición Imagine Cup '09, en su modalidad de Robótica y Algoritmos. Durante este trabajo se realizarán tareas de adquisición de conocimiento, pues la herramienta de programación MRDS es completamente desconocida por el estudiante, así como de investigación del estado actual del arte en el problema presentado. Por otro lado, se comentará detalladamente la solución elaborada para participar en dicha competición y los resultados obtenidos. Esta descripción del proceso servirá como ejemplo para ilustrar las distintas fases por las que estudiantes y entusiastas de distintas disciplinas deben transitar para participar en competiciones, nacionales e internacionales, de similar relevancia.

En esta sección se detallan los objetivos que se persiguen con el desarrollo del proyecto. Dichos objetivos componen el enfoque primario de todas y cada una de las actividades a realizar y, por tanto, su cumplimiento resultará imprescindible para la satisfactoria finalización del proyecto. Estos objetivos se corresponden con algunas de las cualidades requeridas a un ingeniero pues, como veremos, incluyen la obtención y asimilación de nuevos conocimientos sobre las últimas tecnologías, así como la aplicación del conocimiento recién adquirido en la elaboración de una solución a un problema determinado.

1.1.- Análisis y comprensión de la tecnología MRDS 2008

Antes de comenzar con ningún tipo de implementación, resultará necesario realizar un proceso de análisis del entorno de programación robótica denominado Microsoft Robotics Developer Studio 2008 (MRDS). La finalidad de dicho análisis será, por una parte, obtener el suficiente conocimiento sobre la tecnología MRDS para desarrollar una aplicación de control de un robot haciendo uso de ella. Deberán conocerse los distintos componentes que integran dicha tecnología, así como realizarse una breve descripción de los más relevantes en cuanto a programación se refiere. Por otra parte, el análisis de la tecnología generará como producto la mencionada descripción de los componentes más relevantes para la programación, que tendrá como finalidad permitir a un usuario ajeno a la tecnología MRDS adquirir todo el conocimiento extraído durante la realización del análisis.

En resumen, se realizará un análisis de la tecnología MRDS con la doble finalidad de extraer conocimiento sobre su utilización y posibilidades, así como de transmitir dicho conocimiento a otra persona para evitar que ésta deba realizar el mismo análisis de nuevo.

1.2.- Participación en la competición Imagine Cup 2009

Tras obtener los conocimientos necesarios sobre la tecnología MRDS como para hacer uso de la misma en la implementación de una aplicación de control de un robot, se elaborará una solución válida para participar en la segunda ronda de la competición estudiantil internacional *Imagine Cup 2009*.

La solución implementada para la competición deberá superar una serie de retos impuestos por el enunciado de la misma. Dichos retos son los siguientes:

- Implementación de un sistema de navegación robótica, en entorno desconocido, a través de una serie de localizaciones, utilizando como soporte para la localización del robot un sistema GPS y los ángulos eulerianos[1] de navegación *Yaw*, *Pitch* y *Roll*.

- Evasión dinámica de obstáculos fijos y prevención de los efectos adversos derivados de la navegación sobre terreno irregular.

- Análisis de la información visual obtenida a través de distintas cámaras para detección de objetos de interés tanto en el entorno local como en grandes distancias durante la navegación.

- Optimización del proceso de análisis de la información obtenida por los sensores para acelerar la toma de decisiones.

Como paso previo a la implementación de la solución se realizará un análisis de las soluciones ya existentes a cada uno de estos problemas. Dichas soluciones se describen en el apartado 2 como parte del estado del arte de la navegación robótica. Posteriormente, se realiza una descripción de las principales características de la herramienta MRDS para el desarrollo de aplicaciones orientadas al control de robots móviles. A continuación, en el capítulo 4, se describe la competición Imagine Cup, introduciendo brevemente su motivación e historia y detallando el desafío que plantea en su edición 2009, pues es al que nos enfrentaremos. En el capítulo 5, se incluye una descripción de la arquitectura inicial que se obtiene tras realizar la instalación del desafío.

---

[1] Más información en http://planning.cs.uiuc.edu/node102.html y http://en.wikipedia.org/wiki/Flight_dynamics

En el capítulo 6, se describe la solución implementada, en la que se intentará incluir las soluciones descritas en el apartado 2 como solución al problema enunciado en la competición, o utilizarlas como base para un nuevo algoritmo, así como realizar, si fuera posible, algún tipo de optimización de las mismas. Además, en este capítulo se indican los resultados obtenidos en la clasificación general de la competición. En el capítulo 7, se redactan las conclusiones extraídas y se comentan posibles vías de desarrollo futuro. A continuación, se incluye una serie de apéndices. En el apéndice A se explica cómo obtener las herramientas necesarias para ejecutar la solución. En el apéndice B se detalla el contenido del CD adjunto a esta memoria. La planificación del proyecto se describe en el apéndice C y, finalmente, se incluye el código implementado en los distintos ficheros como apéndice D.

Capítulo 2:
Estado de la cuestión
Navegación Robótica

En este capítulo se incluye una descripción del estado actual de los problemas derivados de la navegación robótica. En éste, se incluyen diferentes técnicas utilizadas actualmente para afrontarla. Además, se introducen las nuevas líneas de investigación que tratan de abordar dichos problemas de navegación robótica desde diferentes perspectivas. La navegación es una de las tareas más desafiantes asociadas a los robots móviles. Para poder afrontarla con éxito, debe asentarse en los tres pilares básicos que la componen:

- **Mapeo**: el robot debe poseer una representación del entorno del robot.

- **Localización**: debe poseer mecanismos para estimar su posición en el entorno.

- **Planificación**: debe ser capaz de decidir qué acciones realizar para conseguir sus objetivos. Con respecto a este

De estos tres retos, la localización ha sido, quizás, sobre la que más investigación ha recaído en los últimos tiempos. Debido a ello, existe una amplia gama de teorías y soluciones para llevarla a cabo, como la representación en un mapa del entorno mediante nubes de puntos (Latombe, 1991), métodos de Montecarlo (Handshin, 1990), etcétera. En este capítulo se describen las alternativas clásicas y modernas más representativas y relevantes. Por otro lado, también existen distintas soluciones que permiten la elaboración de planes de acción, con los que el robot decidirá sus siguientes pasos a tomar. Dichos planes, como veremos adelante, se elaboran tras obtener los mapas del entorno en el que el robot se encuentra el robot (mapeo) y una estimación de la posición de este dentro del mismo (localización). Este proceso de planificación puede dividirse, a su vez, en un problema de navegación global, en el que el robot debe planificar una manera de desplazarse entre dos puntos y una navegación local, en la que el robot debe aplicar mecanismos diseñados para permitirle seguir la ruta planificada, como la evasión de obstáculos o la corrección de desvíos en el recorrido.

En este capítulo, por el especial interés que posee para el futuro trabajo a realizar, se describirán distintas técnicas de mapeo, localización y planificación en robótica móvil. Además, se describe una filosofía denominada localización y mapeo simultáneos (Leonard y otros, 1991), que implica realizar estas dos tareas a la misma vez. Por último se incluye la descripción del algoritmo Wedgebug (Laubach y otros, 1999), relacionado, en gran medida, con el problema a resolver.

2.1.- Mapeo

En un problema típico de navegación robótica, se requiere que el robot se desplace desde una localización *A* hasta otra localización *B*, lo que representa un ejemplo de navegación global. Para realizar esta tarea, resulta obvio que se necesitan sensores, para evitar colisiones con objetos del entorno o determinar si se ha alcanzado la posición destino, además de mecanismos de control del movimiento que permitan el desplazamiento del robot en la dirección deseada.

Supongamos que el robot se encuentra en un entorno como el que se muestra en la siguiente figura:

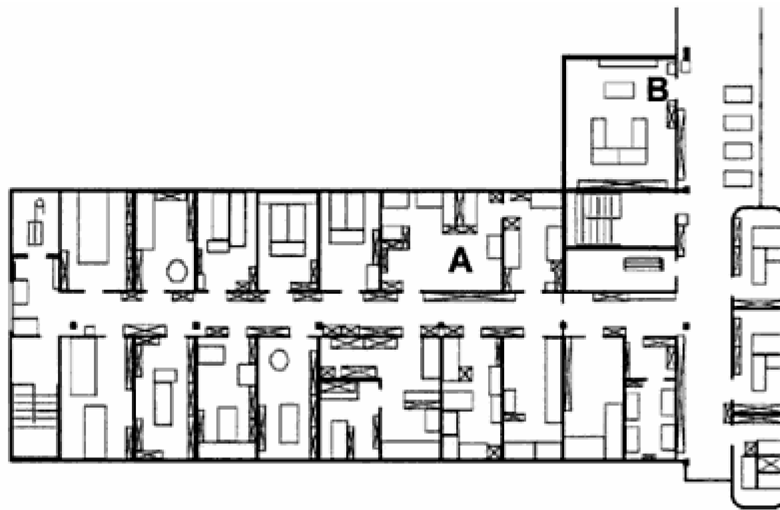

Figura 1.- Representación bidimensional de un escenario de interior

Si deseamos que el robot se desplace desde el punto A al punto B, podrían diseñarse distintos tipos de comportamiento. Por ejemplo, si se implementase una conducta en la que el robot siguiese la pared situada a uno de sus laterales, izquierdo o derecho, acabaría alcanzando el objetivo deseado. Esta solución se corresponde con una aproximación basada en comportamientos. Sin embargo, la solución a esta tarea mediante una aproximación basada en comportamientos consumiría mucho tiempo, que se vería incrementado proporcionalmente al tamaño del entorno. Así pues, se diseñaron otros tipos de soluciones basadas en la obtención de mapas del entorno.

La navegación basada en mapas incluye los módulos de localización y planificación. En esta modalidad de navegación, el robot trata de localizar su posición, con respecto a un mapa del entorno que posee, a través de la información obtenida por sus diferentes sensores. Esta aproximación requiere más esfuerzo por parte del programador, pero su objetivo es permitir que el robot pueda desplazarse a través de distintos escenarios de una manera más eficiente, por lo que dicho aumento del esfuerzo espera verse compensado con el tiempo.

En lo que respecta a la navegación basada en mapas, es necesario tener en cuenta que el robot confía completamente en una representación interna que éste posee acerca del entorno. Dicha representación, ya sea construida por el robot o proporcionada por el programador, puede diferir del entorno real, por lo que resulta posible que el comportamiento del robot no fuese el deseado. En cuanto a los posibles sistemas de representación de un determinado escenario, existen diversas alternativas. No existe ningún tipo de representación que deba usarse en cualquier circunstancia, pues deben tenerse en cuenta una serie de características a la hora de elegir el tipo de mapa y nivel de detalle del mismo que serán utilizados:

- La precisión del mapa debe concordar con la precisión con la que se desea alcanzar un determinado objetivo, pues no se requiere la misma precisión para conseguir que el robot se desplace a la habitación o área B que para conseguir que se desplace al punto B, definido por las coordenadas (X, Y, Z).

- La precisión del mapa y el tipo de elementos representados deben asemejarse a los tipos de información obtenidos por el robot a través de sus sensores. Por ejemplo, no sería apropiado incluir en la representación del entorno el contorno de un elemento indicando que su color es rojo si los sensores sólo pueden identificar objetos y no colores.

- La complejidad en la representación del mapa aumenta proporcionalmente la complejidad computacional requerida para efectuar los procesos de localización y navegación.

Por tanto, a la hora de elegir un tipo determinado de representación, deben conocerse muy bien el tipo de tareas que deberán ser realizadas, así como las implicaciones que derivan de la elección de un tipo de representación u otra.

A continuación, se describen brevemente distintos tipos de representaciones del entorno utilizando mapas.

2.1.1 Representaciones continuas

Los mapas basados en representaciones continuas del entorno permiten obtener una descripción exacta de la composición del escenario en el que se encuentra el robot. En la actualidad, este tipo de representaciones están realizadas en dos dimensiones, pues incluir una tercera supondría un incremento exponencial de la carga computacional requerida.

La aproximación más común consiste en representar en el mapa sólo los objetos que pueden ser detectados por los sensores del robot (Lazanas y otros, 1992), como paredes u obstáculos, y asumir que el resto del espacio está vacío, con lo que el robot puede transitar a través de él. Debido a esta suposición, la cantidad de información que debe ser almacenada disminuye, por lo que se minimizan los requisitos relacionados con la memoria del robot.

Un ejemplo de este tipo de representación puede observarse en la siguiente figura:

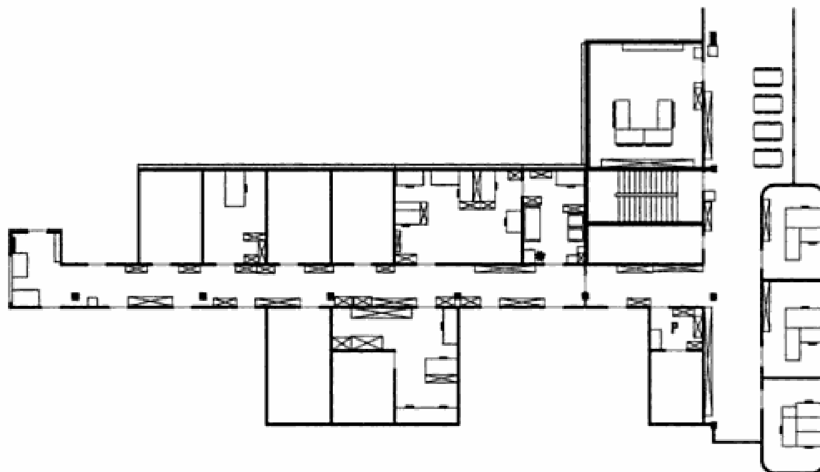

Figura 2.- Representación bidimensional simplificada de un entorno.

En este caso, todos los objetos del escenario se han representado utilizando polígonos geométricos localizados en determinadas coordenadas del espacio. Además, se ha obviado toda la información referente a texturas o colores de dichos objetos, pues el escenario está ideado para que pueda navegar por él un robot haciendo uso sólo de sensores de proximidad o telémetros, como láseres y sónares.

Por otro lado, en este tipo de mapas geométricos se pueden realizar abstracciones aún más profundas que la omisión de características físicas de los objetos, no detectables por el robot. Estas abstracciones incluyen la representación de los elementos mediante polígonos convexos muy simples. Un ejemplo de dicha abstracción, puede obtenerse mediante el método de la representación mediante líneas infinitas (Siegwart y otros, 2004). Este método se fundamenta en la suposición de que la gran mayoría de la navegación robótica en interiores se realiza utilizando sensores de proximidad de tipo láser. Mediante este método, el entorno se representa mediante una serie de líneas infinitas situadas en el espacio. Estas líneas pueden estar situadas en cualquier posición del espacio bidimensional y formando cualquier ángulo con el eje de ordenadas. Una vez obtenidas estas líneas, se incluyen únicamente las que representan elementos que podrían ser detectados por el sensor láser.

La siguiente figura muestra cómo podría representarse un escenario sencillo mediante el método de las líneas infinitas:

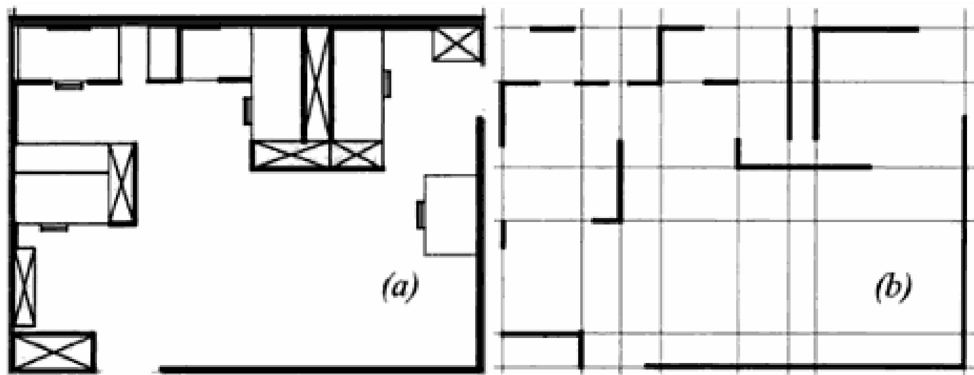

Figura 3.- Representación en líneas infinitas (b) del entorno (a)

Como puede extraerse, esta representación requiere muy poca capacidad de almacenamiento para contener el mapa, lo que conlleva una disminución en la carga computacional necesario para realizar tareas de navegación y localización.

En resumen, los métodos de representación continua poseen la ventaja de permitir una representación fidedigna del entorno que, en conjunto con las técnicas de abstracción mencionadas, proporcionan una manera eficaz de conseguir las bases sobre las que se sustentan las técnicas de localización y navegación.

2.1.2.- Estrategias de descomposición

Como ya comentamos en el apartado anterior, la abstracción de las características del entorno proporciona mejoras en la eficiencia de los procesos de localización y navegación. Por tanto, existen ciertos métodos que aplican dicha abstracción en proporción extrema. Para ello, utilizan técnicas de descomposición del espacio continuo y lo transformar en un conjunto discreto de posiciones. Estas técnicas poseen la desventaja de disminuir la fidelidad del mapa obtenido. Sin embargo, pueden resultar útiles si la abstracción permite capturar sólo las características más relevantes del entorno, obviando las demás, simplificando en gran medida la carga computacional necesaria durante los procesos de localización y, especialmente, de planificación con respecto a métodos de representación continua. Una de estas técnicas de descomposición, es la denominada *Descomposición exacta en celdas* (Latombe, 1991). Mediante esta técnica, el escenario se divide en distintas celdas que representan áreas vacías. Los límites de cada celda son almacenados en diferentes nodos, lo que permite construir un mapa en el que se refleja la interconexión entre las distintas áreas.

En la siguiente imagen se muestra un ejemplo de mapa de entorno utilizando la técnica de descomposición exacta en celdas:

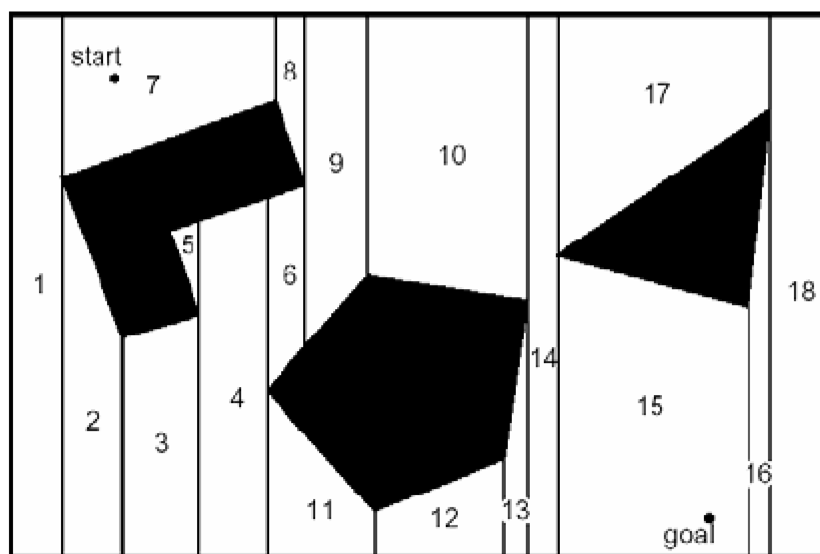

Figura 4.- Descomposición exacta en celdas de un escenario

En este tipo de mapas, la localización exacta del robot dentro de cada una de las celdas es irrelevante, importando únicamente la habilidad del robot para transitar entre celdas adyacentes. Sin embargo, esta aproximación no es siempre factible, pues en escenarios complejos resulta muy costosa la determinación de celdas y, por consiguiente, la elaboración del mapa. Por ello, existen distintas alternativas, como la *Descomposición fija* (Latombe, 1991), en la que el escenario se divide en una serie discreta de áreas del mismo tamaño.

En esta representación, cada área aparece ocupada o vacía dependiendo de si en alguno de los puntos que engloba existe algún obstáculo. En las siguientes imágenes podemos observar un ejemplo de esta alternativa:

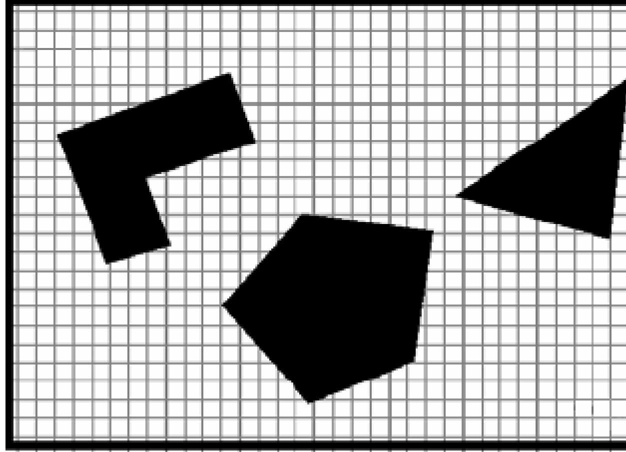

Figura 5.- Descomposición original en celdas

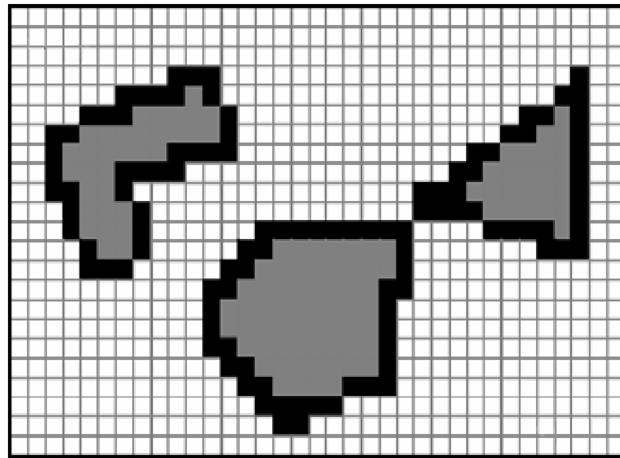

Figura 6.- Descomposición fija

Como podemos observar, existe un pequeño pasaje entre el rectángulo y el pentágono pero, debido a la proximidad entre ambas figuras, en este método se muestra como inexistente. Este caso ilustra perfectamente la principal desventaja del método, que se corresponde con la inexactitud en el mapa generado con respecto al mundo real. Para intentar subsanar el problema de la desaparición de los pasajes diminutos, existe una variante denominada *Descomposición adaptativa (*Latombe*, 1991)*, en la que el tamaño de las áreas es variable. Para obtener las celdas, se obtiene, inicialmente, una única celda que representa a todo el entorno. Inmediatamente después, se divide la celda en cuatro celdas iguales. En cada celda en la que se encuentre un obstáculo se vuelve a practicar la división en cuatro partes iguales, mientras que el resto se mantienen intactas. En las celdas nuevamente generadas se comprueba si existen obstáculos y, en caso afirmativo, se vuelve a realizar una división. Este proceso continúa hasta que no se generen nuevas celdas o hasta que se alcance una determinada precisión.

Un ejemplo de este tipo de descomposición puede observarse en la figura que se muestra a continuación:

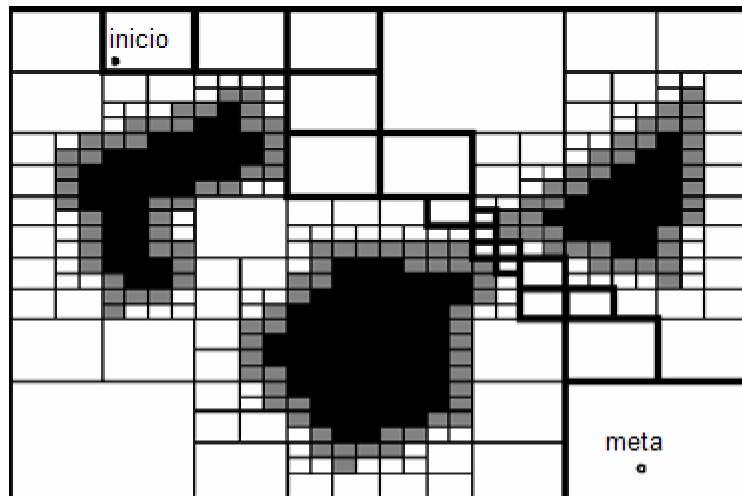

Figura 7.- Descomposición adaptativa

Podemos comprobar que el pequeño pasaje entre figuras vuelve a encontrarse representado.

En la actualidad, el método más utilizado en el campo de la robótica móvil es el denominado *Rejilla de ocupación* (Moravec, 1985). En éste, se posee un entramado de celdas que, en un principio, están marcadas como desocupadas. Cada una de estas celdas contiene un contador que se ve aumentado cada vez que el robot detecta a través de sus sensores un obstáculo localizado en el interior de la misma. Cuando el contador de una celda supera un umbral determinado, se marca esa casilla como ocupada y se representa en un color más oscuro. Por otro lado, a medida que transcurre el tiempo, el contador de las celdas va disminuyendo, con lo que es posible que una celda ocupada volviese a marcarse como vacía. Debido a esto, sería posible representar objetos que se encuentran en el escenario tan sólo un intervalo de tiempo determinado.

Un ejemplo de mapa obtenido mediante esta técnica se muestra en la siguiente imagen:

Figura 8.- Rejilla de ocupación

Los anteriores métodos asumen una representación geométrica del entorno, en la que se posee un escenario con una determinada geometría que no tiene porqué coincidir con la real. Por ejemplo, una sala redonda se representaría en un entramado rectangular existiendo celdas que ocupan posiciones externas a la misma, incrementando la complejidad de la representación de manera innecesaria. Por otro lado, también es remarcable el aumento de capacidad de almacenamiento necesario para contener todas y cada una de las celdas, lo que, como se ha comentado anteriormente, incrementa la complejidad computacional de los algoritmos de localización y planificación.

Por tanto, existe una alternativa de representación a los mapas geométricos, los denominados mapas topológicos (Siegwart y otros, 2004). En éstos, el entorno se encuentra representado como un grafo de conectividad. Cada nodo del grafo representa un área del espacio, mientras que los arcos relacionan los nodos que representan áreas adyacentes en el espacio, es decir, áreas entre las que es posible transitar directamente.

Un ejemplo de mapa topológico puede observarse en la siguiente figura:

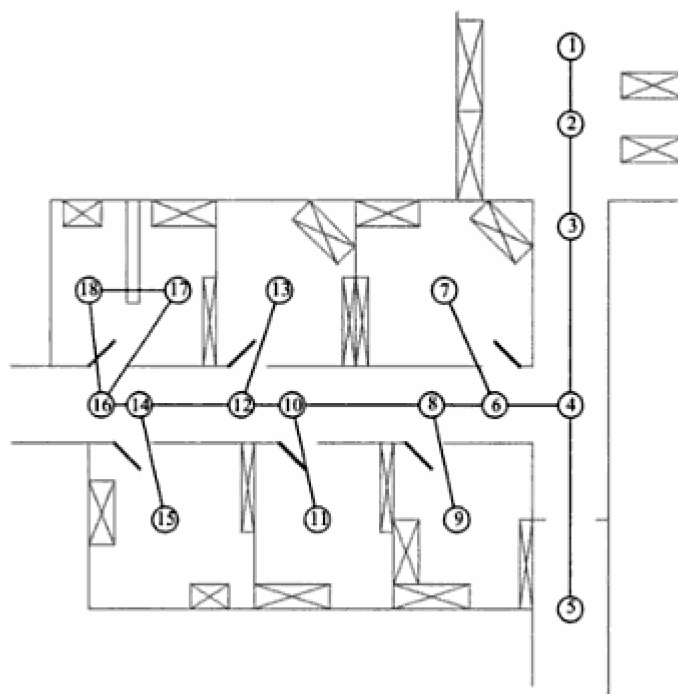

Figura 9.- Mapa topológico

En los mapas topológicos, las áreas representadas por los nodos no tienen porqué ser del mismo tamaño. Sin embargo, resulta necesario que en cada una de las áreas exista cierto elemento discriminante que pueda ser percibido por los sensores y permita al robot identificar el área en el que se encuentra. Así pues, para que un robot pueda navegar a través del espacio utilizando un mapa topológico, deben cumplirse dos características: el robot debe ser capaz de determinar el nodo del grafo en el que se encuentra y debe poseer algún mecanismo para transitar eficazmente entre dos nodos interconectados.

2.1.3.- Retos actuales y nuevas líneas de investigación

Como hemos visto en los anteriores capítulos, la representación que el robot posee del entorno es estática, excepto para el caso de la rejilla de ocupación en la que era posible representar, en cierta medida, un entorno dinámico. Sin embargo, el mundo real no es estático, por lo que en la actualidad se investigan nuevos métodos de representación del entorno. Dentro de esta área de investigación se desarrollan, con frecuencia cada vez mayor, soluciones encaminadas a la utilización de información visual en las tareas de representación del entorno, así como de localización y navegación dentro del mismo.

Con respecto a la representación y localización dentro del entorno, quizás el área de investigación más novedosa corresponde con la fusión sensorial (Crowley, 1993). Mediante técnicas de fusión sensorial se intenta conseguir obtener información sobre el entorno combinando información obtenida a través de diferentes tipos de sensores. Por tanto, la información obtenida no puede ser individualmente obtenida por un único tipo de sensor, sino que resulta de la correcta combinación de los mismos, como ocurre en el caso de las personas. Así pues, cabe esperar que en próximas fechas se consigan grandes avances en el área de la robótica relacionada con la representación de entornos dinámicos.

2.2.- Localización

Mediante el uso de las técnicas comentadas en los capítulos anteriores, es posible obtener una representación aproximada del entorno que rodea al robot. Sin embargo, resulta necesario poseer mecanismos de localización que permitan conocer, con determinada precisión, el punto en el que se encuentra el robot dentro del mapa. En el caso de que la posición inicial del robot en el mapa sea conocida, existe la posibilidad de conocer su nueva posición tras realizar un desplazamiento mediante el uso de la odometría. Este proceso consiste en contener una serie de sensores propioceptivos, es decir, de sensores que perciben información interna al mismo robot, y que informen de los movimientos realizados. Sin embargo, esta información es poco fiable debido a los errores frecuentes y acumulables que se producen tanto en los sensores como en los actuadores, además de posibles agentes externos que desplacen al robot sin su consentimiento, como desniveles en el terreno, fuerzas inerciales o deslizamiento de las ruedas sin desplazamiento. Por tanto, otra serie de mecanismos para localizar al robot dentro del mapa son necesarios.

En lo que respecta al conocimiento por parte del robot de su posición dentro del mapa, existen diversas posibilidades. La primera de ellas, denominada *creencia en hipótesis única (*Siegwart y otros*,* 2004*)*, implica que el robot alberga la opinión de que está situado en una posición determinada del mapa, con cierto grado de precisión.

La principal ventaja de esta variante de representación de la localización proviene del hecho de que, ya que existe una sola hipótesis, no hay lugar para la ambigüedad. Además, dicha falta de ambigüedad en la representación facilita en gran medida la toma de decisiones, pues sólo ha de tenerse en cuenta una posición como punto de partida para la siguiente acción a realizar. De manera similar a la simplificación obtenida en la toma de decisiones, resulta más sencillo identificar la nueva posición del robot al realizar un desplazamiento, pues sólo ha de identificarse una única nueva posición.

En la siguiente serie de figuras se muestran ejemplos de localización del robot en distintos tipos de mapas del entorno utilizando la filosofía de hipótesis única.

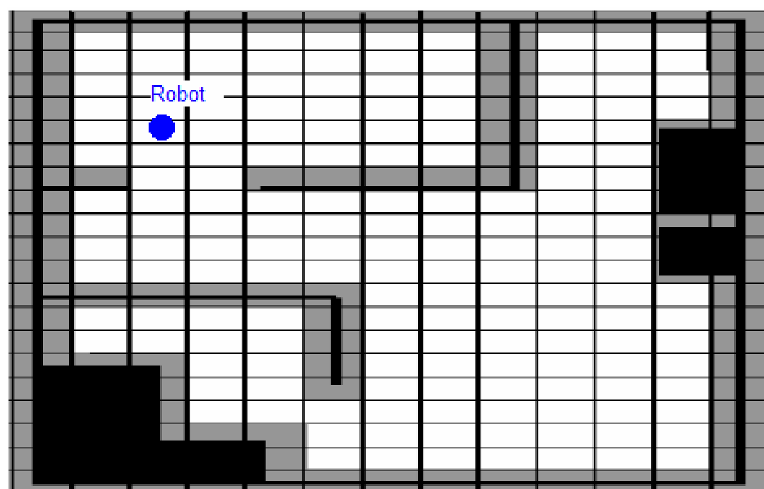

Figura 10.- Localización en mapa de celdas

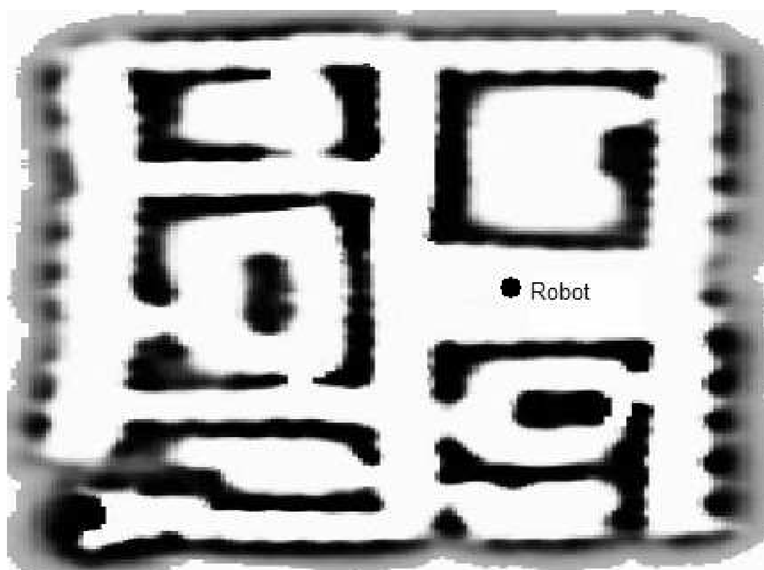

Figura 11.- Localización en rejilla de ocupación

Por otro lado, existe la denominada *creencia en múltiples hipótesis* (Siegwart y otros, 2004), mediante la cual el robot posee diferentes opiniones sobre su posición actual. Para representar este tipo de creencia en un mapa del entorno existen diferentes posibilidades. Una de ellas fue ideada por el investigador Jean-Claude Latombe (Latombe, 1991) y representa el conjunto de posibles posiciones del robot como un polígono convexo en un mapa bidimensional del entorno. Esta representación implica que todas las posiciones contenidas dentro del polígono sean posibles posiciones del robot. De esta manera, el conjunto de posiciones se representa geométricamente dentro del mapa, sin poseer ningún grado de preferencia por ninguna de las posiciones. Es decir, el robot asigna la misma probabilidad a todas las posibles posiciones.

Otro ejemplo de dicha distribución equiprobable lo representa la representación en forma de nube de puntos. Dicha representación se muestra en la siguientes imágenes.

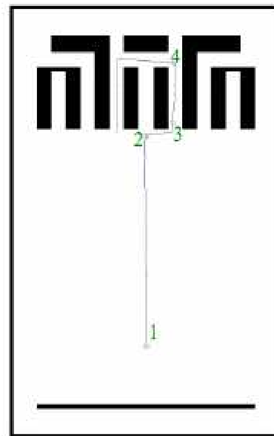

Figura 12.- Camino recorrido

Figura 13.- Nube de puntos en posiciones 2, 3 y 4

Este tipo de representaciones de nubes de puntos equiprobables ha demostrado cierta eficacia, pero resulta útil añadir algún tipo de orden de preferencia entre las distintas posiciones, pues es posible determinar que algunas posiciones son más probables que otras. Para ello, existe la posibilidad de representar el conjunto de posibles posiciones utilizando una expresión matemática.

Por ejemplo, puede definirse una posible posición utilizando sus coordenadas P = (X, Y) acompañada de una probabilidad media μ y una desviación estándar σ, es decir, utilizando una distribución Gaussiana (Thrun y otros, 1998). De esta manera, se representaría la incertidumbre que el robot posee sobre cada una de sus hipótesis. Por otro lado, también sería posible almacenar una serie finita de puntos discretos que representasen todas y cada una de las posibles posiciones del robot, también conocidas como hipótesis. Además, a cada una de estas posiciones podría asociársele un grado de probabilidad o confianza en su exactitud. Así pues, existen diferentes alternativas para representar el grado de confianza del robot en cada una de sus posibles localizaciones. De esta manera se consigue que el robot posea una medida explícita de la incertidumbre respecto a su posición, lo cual no podía conseguirse utilizando la creencia en hipótesis única. Este hecho, por tanto, representa la principal ventaja de esta filosofía.

En lo concerniente a la posterior fase de planificación, el inconveniente fundamental de poseer diferentes hipótesis consiste en el aumento de la complejidad a la hora de tomar decisiones con respecto a la siguiente acción a realizar, pues existe cierta ambigüedad con respecto a la posición inicial desde la que se inicia el movimiento. Para solventar esta situación desfavorable, existen distintas alternativas que varían desde limitar el número total de hipótesis posibles que pueden albergarse, hasta aplicar distintos métodos probabilísticos que permitan simplificar el proceso de actualización de hipótesis.

En este apartado se describen dos métodos probabilísticos frecuentemente utilizados en procesos de localización: el *filtro de Kalman* (Negenborn, 2003) y la *localización de Markov* (Fox y otros, 1999). Estos métodos requieren la elaboración previa de un mapa del entorno para poder realizar su proceso de localización.

2.2.1.- Localización de Markov

Para poder aplicar este método, es preciso que el robot represente las hipótesis sobre su posición utilizando una función probabilística. Además, en la práctica se precisa obtener un conjunto discreto de posibles posiciones antes de comenzar con la aplicación del método. En la actualidad, dicho conjunto está formado por un número de posiciones que oscila entre cientos y millones. Por tanto, se requiera un mecanismo eficiente para actualizar el conjunto de hipótesis tras realizar algún desplazamiento. Dicha actualización consiste en añadir y eliminar nuevas posiciones al conjunto, si fuera preciso, y actualizar la probabilidad marginal, $p(X)$, asociada a cada una de las posiciones, en base a la información percibida a través de los sensores. Cada una de estas posiciones se representa dentro del mapa indicando sus coordenadas cartesianas (x,y) y su rotación r con respecto a uno de los ejes del robot. Así pues, cada localización l perteneciente al espacio L se representa de la siguiente manera:

$$l = (x, y, r).$$

En un determinado instante de tiempo, *t*, la creencia del robot sobre cada una de sus posibles soluciones está determinada por una función de probabilidad $Bel(L_t)$. Así pues, la probabilidad asociada a cada una de las hipótesis en un instante determinado se denota $Bel(l_t)$ .Así pues, el método consiste en actualizar la distribución de probabilidad de cada una de las posibles localizaciones a lo largo del tiempo, cada vez que se realiza un desplazamiento o varía la información percibida a través de sus sensores.

Dependiendo de la causa de la que derive la aplicación del método, se aplicará el modelo de movimiento o el modelo de percepción del robot:

- El **modelo de movimiento** se aplica cada vez que el robot realiza un desplazamiento. La magnitud, que incluye componentes de rotación y traslación, de dicho desplazamiento *d* se percibe a través los sistemas de odometría. Por tanto, para calcular la distribución de probabilidad *Bel(l')* para cada una de las hipótesis *Bel(l),* es decir, la probabilidad de que el robot se encuentre en la posición *l'* tras realizar un desplazamiento *d* desde la posible posición *l* y que se denota *p(l'|l,d)*, es necesario integrar el producto de ambas probabilidades a lo largo del espacio *L*:

$$Bel(l') = \int_L p(l'|l,d) \cdot Bel(l) \cdot dl$$

- El **modelo de percepción** se aplica cada vez que varía la información obtenida por los sensores. Este modelo utiliza la función de probabilidad *p(s|l)* que denota la probabilidad de que se perciba una lectura sensorial *s* desde una determinada posición *l*. Esta probabilidad se obtiene realizando un análisis del mapa del entorno. Así pues, al aplicar este modelo, se actualizan las distribuciones de probabilidad asociadas a cada una de las hipótesis de la siguiente manera:

$$Bel(l_t) = p(s|l) \cdot Bel(l_{t-1})$$

En la siguiente imagen se ejemplifican los resultados obtenidos al aplicar el método de localización de Markov desde una posición inicial desconocida:

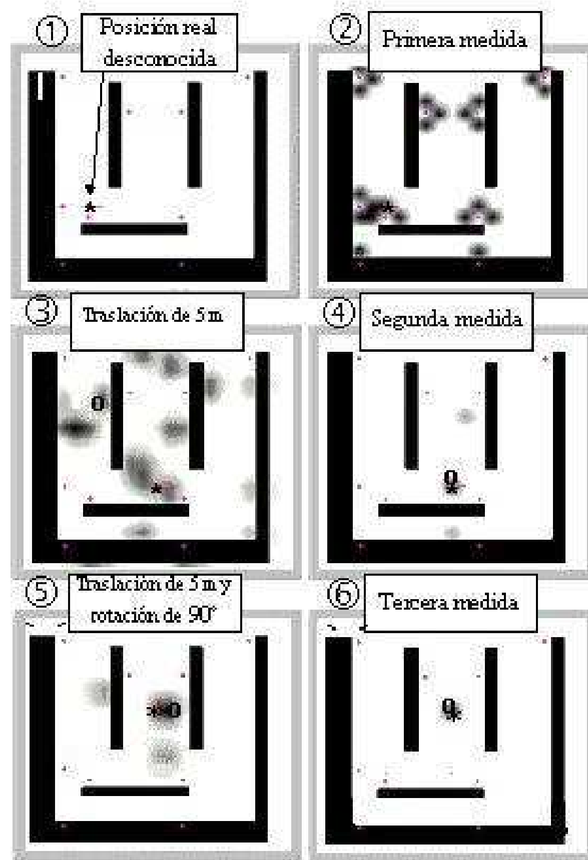

Figura 14.- Proceso de localización de Markov

Como podemos observar en (1), la posición inicial del robot dentro del mapa del entorno es desconocida. Por tanto, se aplica el método de la percepción sobre todo el espacio L, tras aplicar cierta discretización al conjunto, y se obtiene una serie de hipótesis, representadas como nubes de partículas en (2). A continuación, el robot realiza un desplazamiento, por lo que aplica el método del desplazamiento (3), así como el de percepción, pues sus lecturas varían. Por tanto, las hipótesis varían, como se muestra en (4). Finalmente, el robot realiza un nuevo desplazamiento, aplicando los métodos de desplazamiento (5) y percepción (6), y consigue determinar su posición dentro del mapa.

Para finalizar, debe remarcarse que, como se ha mostrado en este capítulo, el método de la localización de Markov posee la ventaja de permitir la localización del robot dentro entorno partiendo de una posición inicial desconocida, lo que puede ser muy útil si el robot sufre errores en sus sistemas de odometría o si es desplazado por la acción de agentes externos.

2.2.2 Filtro de Kalman

El filtro de Kalman es una técnica de fusión sensorial. Por tanto, para aplicarlo en procesos de localización, dichos problemas de localización deben representarse como problemas de fusión sensorial. Esta técnica consta de diversas fases en las que se adquieren distintos tipos de información, elaborando una serie de predicciones, y concluye con una fase en la que se realiza un proceso de emparejamiento entre los distintos tipos de predicciones realizadas.

En la aplicación de este procedimiento, se realiza una estimación inicial de la posición del robot en base a la posición del mismo antes de realizar un movimiento y los valores odométricos obtenidos al realizarlo. Por tanto, para realizar una estimación de la posición tras realizar el primer movimiento, es necesario conocer, con determinada confianza, el punto inicial de partida del robot. Tras realizar la predicción inicial de la posición, se obtienen diferentes lecturas del entorno a través de los sensores. Cada una de estas lecturas posee diferente información sobre el entorno, lo que permite realizar nuevas predicciones de la posición del robot dentro del mapa. Finalmente, se poseen distintos conjuntos de posibles localizaciones, por lo que se realiza un proceso de fusión sensorial en el que se intenta minimizar el error en la predicción, derivado de errores en las distintas mediciones, a través de la extracción de características comunes a los distintos conjuntos de predicción. Para que dicha minimización del error pueda realizarse, se asume que el error en las distintas mediciones puede evaluarse mediante una distribución gaussiana, hecho que no es siempre cierto, pero que, no obstante, ha permitido obtener buenos resultados.

Por otro lado, también se asume que las relaciones entre la posición y las medidas de los distintos sensores tienen carácter lineal, por lo que no podría utilizarse este procedimiento en un sistema no lineal. Sin embargo, se han desarrollado distintas modificaciones, como el Filtro de Kalman Extendido, en las que no es necesario aplicar estas restricciones al problema.

2.3.- Localización y Mapeo Simultáneo: SLAM

Esta filosofía consiste en la elaboración dinámica de un mapa del entorno por parte del robot, a medida que se desplaza por el escenario, en base a la información obtenida a través de sus sensores, efectuada conjuntamente con el proceso de localización del robot dentro del mismo. Es decir, mediante este tipo de procedimientos, el robot elabora un mapa de su entorno al mismo tiempo que localiza su posición actual dentro del mismo.

SLAM no es un algoritmo propiamente dicho, sino que se compone de una serie de subtareas que pueden ser realizadas de distinta manera. Es, pues, una combinación de técnicas de mapeo y localización como las incluidas en los capítulos anteriores. Entre estas tareas se encuentra la localización de marcas del entorno, fusión sensorial, estimación del estado y actualización del estado y marcas del entorno. El primer paso que el robot realiza consiste en detectar ciertos elementos del entorno que puedan ser utilizadas como marcas de posición. Estas marcas pueden estar predefinidas de antemano, con lo que el robot intenta localizarlas, o ser seleccionadas dinámicamente. Para ello, el robot realiza distintas observaciones a través de sus sensores e identifica elementos que pueden ser reobservados en sucesivas ocasiones. Una vez que las marcas de terreno están identificadas, el robot las incluye en el interior de un mapa del entorno que está elaborando. Además, en ese mismo instante el robot actualiza su posición o estado dentro del mencionado mapa. Tras realizar un movimiento, el robot actualiza su posición dentro del mapa utilizando técnicas de odometría. Sin embargo, dichas técnicas conllevan cierto error asociado que puede acumularse a lo largo de distintas iteraciones, ocasionando que tanto la elaboración del mapa del entorno, como el posicionamiento del robot dentro del mismo, no concuerden con la realidad. Así pues, tras realizar un proceso de actualización de la posición, el robot vuelve a realizar una observación del entorno en busca de las marcas de terreno identificadas previamente. Esta observación puede ser realizada utilizando diferentes tipos de sensores, por lo que se obtendrán diferentes tipos de información que conlleva, a su vez, cierto grado error en la medición.

Una vez que se ha obtenido la información a través de los distintos sensores, resulta necesario realizar un proceso de fusión sensorial con el objetivo de obtener la aproximación más real de la posición de las marcas observadas. Para este proceso, resulta frecuente utilizar una variante del Filtro de Kalman denominada Filtro de Kalman Extendido.

Tras obtener la nueva posición del robot y las marcas de terreno, se actualiza tanto el mapa interno que el robot elabora como su posición en el mismo. De esta manera, el robot no precisa ningún tipo de información previa al comienzo de la navegación, sino que puede comprender, en cierta medida, de manera autónoma el entorno que le rodea y el lugar que él mismo ocupa.

En la siguiente imagen se muestra un mapa del entorno que no ha sido elaborado previamente al proceso de navegación, sino que se ha obtenido mediante una técnica SLAM:

Figura 15.- Mapa del entorno obtenido mediante una técnica SLAM

Además, también se puede apreciar, en un trazo más tenue, la trayectoria seguida por el robot durante la elaboración.

2.4.- Planificación

En los anteriores capítulos se han descrito diversas técnicas mediante las cuales es posible que el robot obtenga una representación del entorno que le rodea (mapeo), así como su posición dentro del mismo (localización). Una vez que el robot posee esa información, resulta necesario que éste posea algún mecanismo mediante el cual pueda decidir qué acciones realizar, en este caso qué desplazamientos llevar cabo, con el fin de cumplir unos determinados objetivos.

Para ello, existen distintas alternativas que permiten generar un plan de acción que garantice, en buena medida, la consecución de sus metas. Para ello, se divide el proceso en dos tareas: la planificación global, mediante la que se elabora un plan para conseguir el desplazamiento del robot entre dos zonas del mapa, y la planificación local, en la que se deciden qué acciones realizar para evitar elementos que interfieren en la ruta planificada, como obstáculos o fuerzas externas. Estas tareas de planificación implican, principalmente, la generación de un camino a seguir por el robot que le permite desplazarse hasta su destino de la manera más rápida y eficaz, además de la evasión de obstáculos presentes en el entorno.

A continuación, se presentan distintos algoritmos de planificación de caminos.

2.4.1.- Grafo de Visibilidad

Esta técnica de generación de caminos se basa en encontrar el camino más corto posible desde la posición en el que el robot se encuentra hasta su destino (Latombe, 1991). Para ello, y dado un mapa geométrico del entorno, genera caminos que unen todos los vértices de los elementos del terreno dos a dos, incluyendo las posiciones inicial y final.

Una vez que esta red de caminos se ha generado, identifica el conjunto de los mismos que forman el camino más corto entre las posiciones inicial y final. El funcionamiento de este método se ilustra en la siguiente imagen:

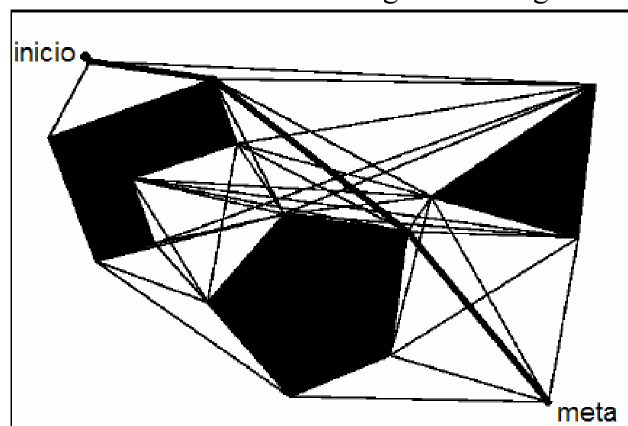

Figura 16.- Grafo de visibilidad

Como podemos observar, en la imagen se muestran todos los caminos que forman la red, así como el camino más corto, en líneas más gruesas, que será el seguido por el robot. Por otro lado, podemos comprobar también que dicho camino, siempre que deba evitar un obstáculo, transitará por un espacio muy próximo a éste.

2.4.2.- Diagrama de Voronoi

De manera contraria al anterior método, mediante el uso de diagramas de Voronoi se genera el camino más alejado posible de los obstáculos del espacio (Latombe, 1991). Para ello, se genera una serie de caminos que equidistan de los obstáculos del terreno.

De esta manera, una vez que se han generado los distintos caminos, se elige la vía más corta a través de ellos que permita al robot alcanzar su posición final. En la siguiente figura se muestra un ejemplo de este proceso:

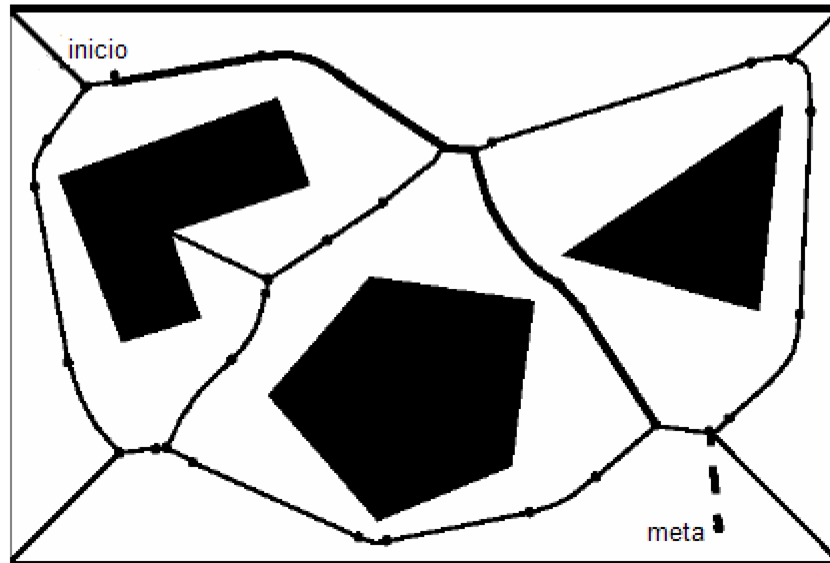

Figura 17.- Diagrama de Voronoi

Como podemos observar, el camino recorrido no es el óptimo en términos de distancia, lo que representa la principal desventaja de este algoritmo. Por otro lado, debido a que es posible que la distancia a los obstáculos sea relativamente grande, cabe la posibilidad de que el camino elegido circule tan alejado de los obstáculos, y otros elementos del entorno, que no sea posible percibirlos a través de los sensores, hecho que dificultaría las tareas de localización.

2.4.3 Método del gradiente

Este método de elaboración de caminos se basa en tratar el mapa del entorno como un campo de fuerzas (Hwang, 1992). Debido a ello, se asigna una fuerza atractiva a la posición destino y fuerzas repulsivas a los obstáculos del mapa.

De esta manera, se genera un campo de fuerzas en el escenario. Entonces, el planificador hará que el robot siga un camino que simule la trayectoria que seguiría una partícula sobre la que ejerciesen las fuerzas de dicho campo. Así pues, el robot se dirigiría desde su posición hasta la meta esquivando todos los obstáculos del escenario. Sin embargo, la simulación del campo de fuerzas suele ser computacionalmente costosa, lo que representa una desventaja para este algoritmo.

En la siguiente figura se muestra el camino elegido por un sistema planificador de un robot utilizando el algoritmo del gradiente.

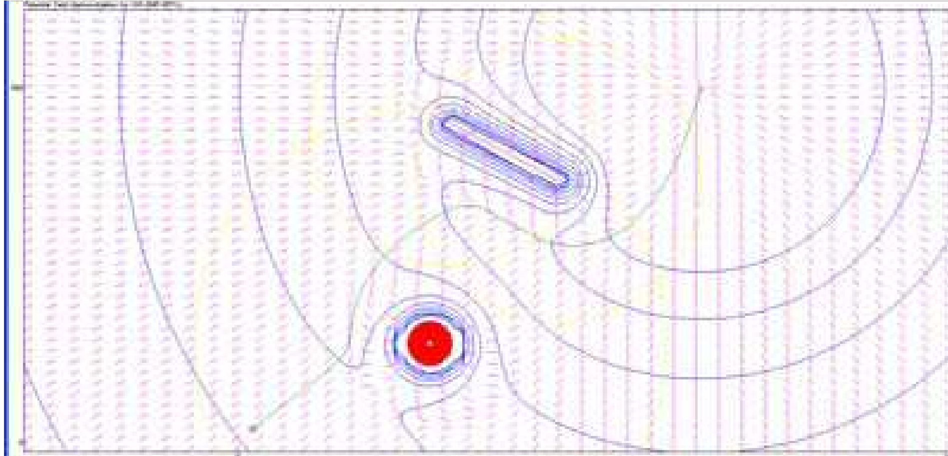

Figura 18.- Método del gradiente

2.5.- El algoritmo Wedgebug

Este algoritmo fue diseñado por científicos del Jet Propulsion Laboratory (JPL) de la Universidad de California, división que realiza investigaciones para la NASA. La razón de su aparición tiene su origen en la necesidad de dotar de una mayor autonomía de navegación a los robots enviados por la NASA a la superficie de Marte. Las premisas básicas de este algoritmo suponen que el robot es sólo un punto dentro de una representación bidimensional del escenario que lo rodea y que todo espacio del entorno está ocupado por un obstáculo o bien está vacío. Por otro lado, los sensores que utiliza permiten al robot rastrear un área en forma de cuña con las siguientes características:

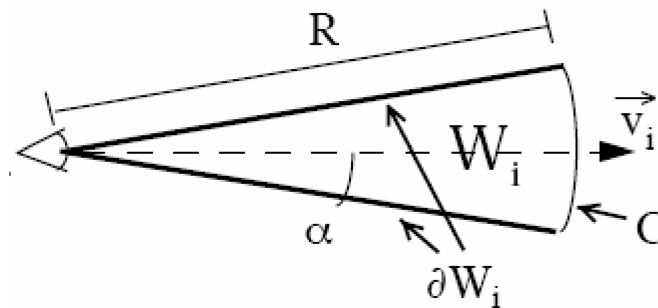

Figura 19.- Descripción del área en forma de cuña analizado

El radio de la cuña es *R* y el sensor barre un área *2α* centrado en la dirección *v*. El arco que delimita la cuña es *C* y el área definida por la cuña es *W*.

Este algoritmo posee dos modos de ejecución denominados *desplazamiento a la meta* y *seguimiento de contorno*. El modo dominante es el de desplazamiento a la meta, pues es mediante el cual el robot se dirige de manera más directa a su objetivo. Para ello, el primer paso que realiza es calcular la distancia que lo separa de su meta, con lo que obtiene una medida de la mayor distancia que debería recorrer mediante este modo.

A continuación, realiza un escaneo en la dirección v0=(posición_actual, destino) y, en ese mismo instante, comienza a elaborar un grafo de visibilidad (LTG) en el que se incluyen los caminos a seguir hasta la meta. Tras realizar este escaneo, pueden darse dos situaciones: que no se detecte ningún obstáculo en el camino, con lo que se añade un nuevo nodo al grafo situado a una distancia R de la posición actual del robot en la recta que une a éste y la meta; o que se detecte un obstáculo, con lo que se añaden tantos nuevos nodos al grafo como puntos de intersección existan entre el área escaneado y el obstáculo.

Un ejemplo de este proceso de extracción de nodos se ilustra en la siguiente imagen:

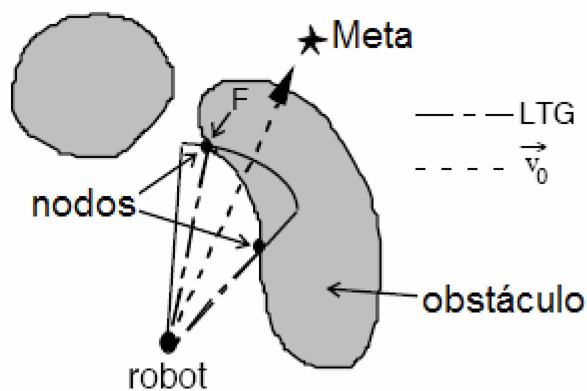

Figura 20.- Extracción de nodos algoritmo Wedgebug

Además, se pueden realizar escaneos adicionales de las áreas adyacentes, con el objetivo de obtener una mejor visión del entorno. Tras obtener la nueva serie de nodos, se ejecuta un algoritmo de camino mínimo que obtiene el subgrafo que más se aproxime a la meta y se desplaza siguiendo ese camino. Una vez que se ha recorrido el camino determinado por el último subgrafo seleccionado, se comienza de nuevo el proceso de escaneo. Este comportamiento se repite hasta que se alcanza la posición destino o hasta que se alcanza un mínimo local, es decir, que cualquier movimiento aumenta la distancia con respecto a la meta. En este último caso, se abandona el modo de desplazamiento a meta y se transita al estado de seguimiento de contorno.

El modo de seguimiento de contorno utiliza, nuevamente, un grafo de visibilidad en el que incluye todos los nodos por los que puede desplazarse. Sin embargo, en este caso, el objetivo del robot no es desplazarse hasta la meta, sino bordear un obstáculo. Para ello, realiza diversos escaneos del obstáculo a bordear en busca de sus límites. Una vez que dichos límites se han localizado, añade los nodos correspondientes a su grafo de visibilidad y ejecuta de nuevo el algoritmo de camino mínimo para desplazarse a uno de ellos.

Este proceso de adquisición de nodos se ilustra en la siguiente imagen:

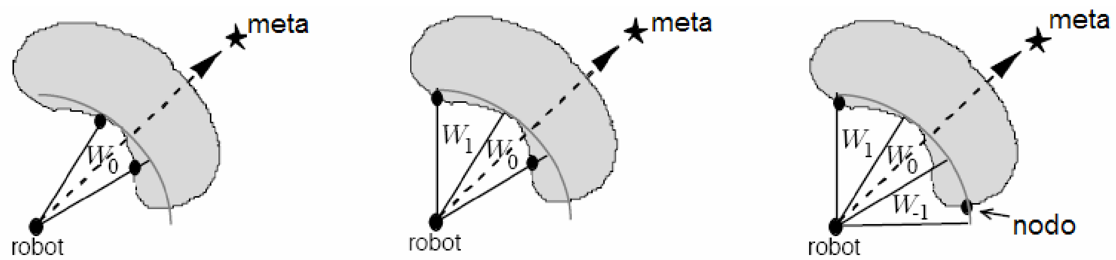

Figura 21.- Proceso de adquisición de nodos mediante diversos escaneos

Este proceso continúa hasta que se detecta un bucle, con lo cual la meta se declara inalcanzable y se detiene el algoritmo, o hasta que se detecta un nuevo camino hacia la meta, una vez bordeado el obstáculo, con lo que se transita de nuevo al modo de desplazamiento a la meta. De esta manera, se consiguió un algoritmo de navegación que permitía que el robot elaborase su propio camino sin necesidad de poseer un mapa del entorno. Además, se dotó de mayor autonomía al robot, pues, anteriormente, todo desplazamiento debía ser coordinado desde la Tierra, lo que disminuía enormemente la eficiencia en la navegación.

Capítulo 3:
Herramientas

Para elaborar una solución al problema, es necesario utilizar las herramientas Microsoft Visual Studio 2008 y Microsoft Robotics Developer Studio 2008. En los siguientes apartados se incluye una breve descripción de las mismas y su utilidad en el desarrollo de una solución robótica.

3.1.-Microsoft Visual Studio 2008

Este entorno de desarrollo integrado será utilizado para crear las soluciones que contengan el comportamiento de los robots. Este entorno permite, entre otros, la programación en lenguaje C#, que será el utilizado para crear el código fuente de nuestra solución.Por otro lado, también es posible compilar el código fuente y realizar una llamada al proceso que activa la ejecución de MRDS, ejecutando así el entorno virtual y el robot que hemos codificado junto con su comportamiento con el fin de testear nuestra solución.

3.2.-Microsoft Robotics Developer Studio 2008

Esta herramienta está diseñada para permitir el desarrollo de aplicaciones orientadas al control de robots. Una de sus principales ventajas es que permite el control de robots físicos y simulados utilizando el mismo código fuente controlador. Esta aplicación funciona de manera similar a un sistema operativo en lo que a la ejecución de los controladores se refiere, pero depende de una plataforma .NET Framework ejecutada sobre un sistema operativo Microsoft Windows para poder ejecutarse, por lo que los robots deben tener una conexión con un ordenador en el que MRDS se esté ejecutando sobre Windows. Se trata de una herramienta muy completa entre cuyas características y posibilidades se encuentran las siguientes:

- Simulación en entornos virtuales en tres dimensiones utilizando la tecnología AGEIA PhysX. Gracias a ésta, se puede simular con alto grado de precisión la dinámica e interacciones físicas entre los elementos del entorno simulado.

- Arquitectura orientada a servicios, gracias al uso de la tecnología Decentralized Software Services (DSS). Ésta permite manejar un robot como un conjunto de servicios que se ejecutan en paralelo y cuya orquestación entre sí resulta clave para definir el comportamiento del mismo.

- Sencillo manejo de la programación en paralelo concurrente gracias a las librerías Concurrency and Coordination Runtime (CCR) que permiten gestionar la concurrencia y coordinación de los anteriormente mencionados servicios. Además, también proporciona sencillas directivas para el control de la programación asíncrona que, como veremos más adelante, resulta vital para poder coordinar los distintos procesos que componen un robot.

- Control de un dispositivo físico a través de una red inalámbrica o directamente desde un robot con PC integrado que incluya una versión actual de Microsoft Windows.

- Proporciona un lenguaje de programación visual denominado Visual Programming Language que resulta bastante intuitivo y ayuda en gran medida al aprendizaje de la programación orientada a servicios utilizando DSS.

Por todas estas características, MRDS se convierte en una muy útil herramienta para la programación robótica. Además de fomentar la investigación en robótica por parte de aficionados sin recursos al estar disponible una versión gratuita de la misma para estudiantes, investigadores y profesores (Academic Edition). Como esta herramienta será utilizada durante el desarrollo de la solución, resulta necesario adquirir cierto nivel de comprensión sobre las características y posibilidades de la misma. Además, dicha adquisición de conocimiento se corresponde con uno de los objetivos del proyecto.

Por tanto, en los siguientes apartados se describen los conocimientos indispensables acerca de los dos módulos principales de la tecnología MRDS 2008: los entornos de ejecución Decentralized Software Services y Concurrency and Coordination Runtime.

3.2.1.- Decentralized Software Services

Este entorno proporciona un modelo de arquitectura orientada a servicios, que requieren baja carga computacional, basados en la tecnología .NET. Dichos servicios siguen un modelo REST (Transferencia de eStado REpresentacional), es decir, su dinámica está basada en los cambios de su estado interior y los mensajes y notificaciones que intercambian entre sí que afectan al mismo. Todos y cada uno de los anteriores servicios precisan de un nodo DSS en el que ser ejecutados. Dichos nodos sirven de entorno de ejecución para los servicios y son los encargados de iniciarlos y detenerlos. El servicio podría considerarse la unidad básica de computación, pues una aplicación puede considerarse formada por un conjunto de servicios que se ejecutan de manera independiente y se comunican a partir de mensajes que atraviesan sus puertos.

Un servicio está formado por diversos componentes, como muestra el siguiente esquema:

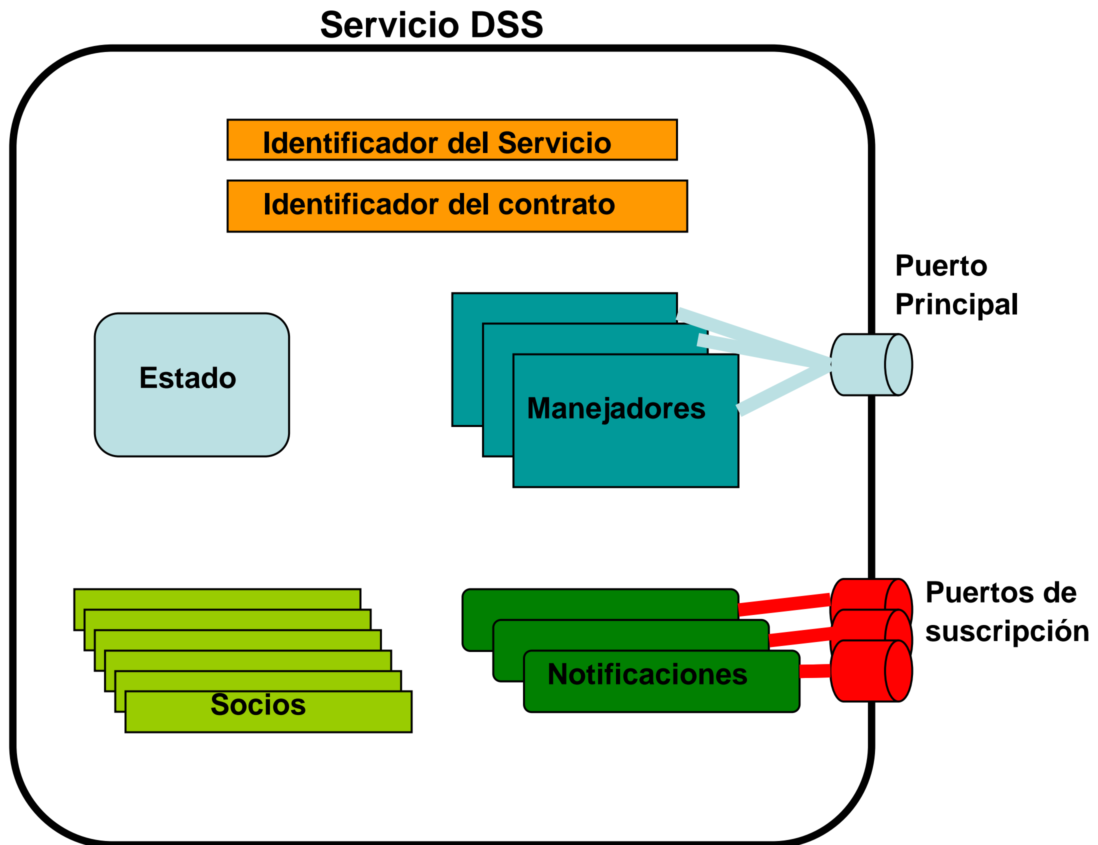

Figura 22.- Servicio DSS

Como podemos observar, los componentes de un servicio DSS son:

- **Identificador del servicio** (Service Identifier): este identificador se crea cada vez que se inicializa una instancia del servicio y representa un identificador universal (uri) para el mismo. Gracias a éste, otros servicios asociados o un navegador web pueden identificarlo y comunicarse con él.

- **Identificador del contrato** (Contract Identifier): se trata de una descripción o identificación única del comportamiento del servicio. En este comportamiento se definen los tipos de operaciones que el servicio acepta, así como los tipos de datos que requieren dichas operaciones.

- **Estado** (State): se trata del estado del servicio en un momento determinado. Este estado está compuesto por cualquier tipo de información que caracteriza al servicio y es accesible, modificable o monitorizable.

- **Socios** (Partners): no es frecuente que un servicio se ejecute en solitario en una aplicación sin relacionarse con ningún otro. Por ello, se registran una serie de servicios socios con los que el servicio podrá comunicarse e interactuar. Durante la inicialización del servicio, se intenta crear una conexión con cada uno de los servicios socios y pueden definirse distintos comportamientos dependiendo de si es posible realizar dicha conexión o no.

- **Manejadores** (Service Handlers): por cada una de las operaciones que soporta el servicio, debe existir un manejador. Este manejador es el encargado de ejecutar una determinada tarea cuando a través del puerto principal llega la solicitud de una determinada funcionalidad. Por cada tipo de operación incluida en la descripción del servicio, exceptuando DSSPDefaultLookup y DSSPDefaultDrop, un manejador debe ser registrado.

- **Puerto principal** (Main Port): se trata de un mecanismo de concurrencia y coordinación perteneciente al módulo CCR (podemos ver aquí lo íntimamente ligados que CCR y DSS están). A través de este puerto el servicio se comunica con el exterior recibiendo mensajes de solicitud de ejecución de operaciones por parte de otros servicios asociados, o de un navegador Web, y emite los mensajes de respuesta oportunos. Es también conocido como puerto de operaciones.

- **Notificaciones** (Notifications): se trata de mensajes de aviso emitidos por el servicio ante cambios en su estado hacia los servicios que se han suscrito al mismo interesándose por sus actualizaciones.

- **Puertos de suscripción**: puertos a través del que los servicios suscriptores reciben las notificaciones ante las actualizaciones de estado ocurridas en el servicio.

En el capítulo 6 se incluye una descripción total del servicio a implementar como solución al desafío de la segunda ronda de la competición de robótica y algoritmos Imagine Cup 2009 y en él se puede observar código de ejemplo para cada uno de estos componentes.

3.2.2.- Concurrency and Coordination Runtime

En la programación robótica debe controlarse una multitud de dispositivos que funcionan al mismo tiempo. Además, como se introdujo en el apartado anterior, una aplicación típica de MRDS está compuesta por multitud de servicios que se ejecutan simultáneamente.

Debido a esto, anteriormente resultaba necesario programar una aplicación en la que los mecanismos de coordinación y sincronización entre los distintos hilos de ejecución estuviesen implementados de forma explícita en el código. Esta necesidad suponía, por un lado, que el programador debía diseñar los mecanismos de concurrencia y coordinación utilizados en la aplicación, añadiendo gran complejidad a la programación, y, por otro, que debía conocer con anterioridad la arquitectura sobre la que iba a ser ejecutada la aplicación. Sin embargo, gracias a CCR es posible abstraer todos estos aspectos, pues se provee un mecanismo para el control de los distintos hilos de ejecución más eficiente que los tradicionales *threads* (hilos) de Windows y mucho más transparente para el programador.

Como se muestra en el apartado anterior, resulta imprescindible poseer un mecanismo para coordinar la comunicación entre los distintos servicios que se ejecutan simultáneamente. Por ello, en este apartado se describen los principales elementos CCR que es necesario conocer para poder crear una solución MRDS válida para controlar un robot.

3.2.2.1.- Puertos

Un elemento de todo servicio es su puerto principal. Este puerto es un mecanismo CCR que implementa una cola FIFO (*First-in*, *First-out*) de mensajes. A este puerto llegan los mensajes que contienen las peticiones, encapsuladas dentro de determinadas clases, que provienen desde otros servicios asociados, y las respuestas generadas por el servicio en contestación a dichas peticiones. Este mecanismo se encuentra implementado en la clase genérica *Port.* Una instancia de esta clase sólo puede recibir mensajes de un determinado tipo, que es especificado durante su instanciación. Por tanto, para poder permitir el envío de distintos tipos de mensajes a través de un mismo puerto principal, se crea la clase *PortSet* que implementa un agregado de colas de distinto tipo que son tratadas como una única entidad.

Gracias a este mecanismo, se puede definir qué tipo de mensajes puede un servicio intercambiar con otros servicios asociados en tiempo de compilación, evitando posteriores errores.

3.2.2.2.- Árbitros y Receptores

En una aplicación MRDS, el envío y recepción de mensajes entre los distintos servicios se realiza de forma asíncrona, por lo que CCR permite declarar árbitros, denominados *Arbiters*, que permiten continuar con la ejecución de otras partes de código mientras se espera a la llegada de mensajes a través de algún puerto.

Estos árbitros coordinan una serie de receptores encargados, a su vez, de observar un determinado puerto a la espera de la llegada de un determinado tipo de mensaje. Tras la llegada de un mensaje, el receptor adecuado para ese tipo de mensaje lo captura y el árbitro indica qué acciones se deben llevar a cabo.

Como ya se comentó en el anterior apartado, estas acciones que se ejecutan tras la llegada al puerto principal de un mensaje de un determinado tipo, se denominan manejadores. Un árbitro puede invocar distintos manejadores dependiendo del tipo del mensaje que llegue al puerto sobre el que actúa. Por otro lado, estos árbitros pueden realizar también complejas operaciones como fruto de la combinación de mensajes capturados por los distintos receptores que coordinan.

3.2.2.3.- Tareas, colas de ejecución y dispensadores

Las aplicaciones MRDS, y las de la robótica actual, en general, se ejecutan en un entorno multitarea. Por ello, es necesario poseer un mecanismo que permita asignar hilos de ejecución a las distintas tareas que han de realizarse. En el caso de CCR este proceso se lleva a cabo por los dispensadores o, en su lengua original, *Dispatchers*. Como vimos en el anterior apartado, es posible continuar la ejecución de ciertas partes de código mientras otras quedan a la espera de que ocurran ciertos eventos, como cuando se habilita un receptor que espera en un puerto. Para ello, el entorno CCR proporciona una interfaz denominada ITask que permite encapsular las distintas funciones posibilitando así su almacenaje en una cola de ejecución a la espera de poder ser ejecutada.

A su vez, existen unos mecanismos de planificación, denominados dispensadores, cuyo cometido es asignar a las distintas tareas almacenadas en la cola de ejecución un hilo de ejecución sobre el que poder ejecutarse. Estos dispensadores contienen un fondo común de hilos de ejecución que poder asignar a las tareas para su ejecución, consiguiendo así un entorno multitarea.

Para aportar una visión global de todo el anterior sistema de coordinación multitarea, podemos observar el siguiente esquema:

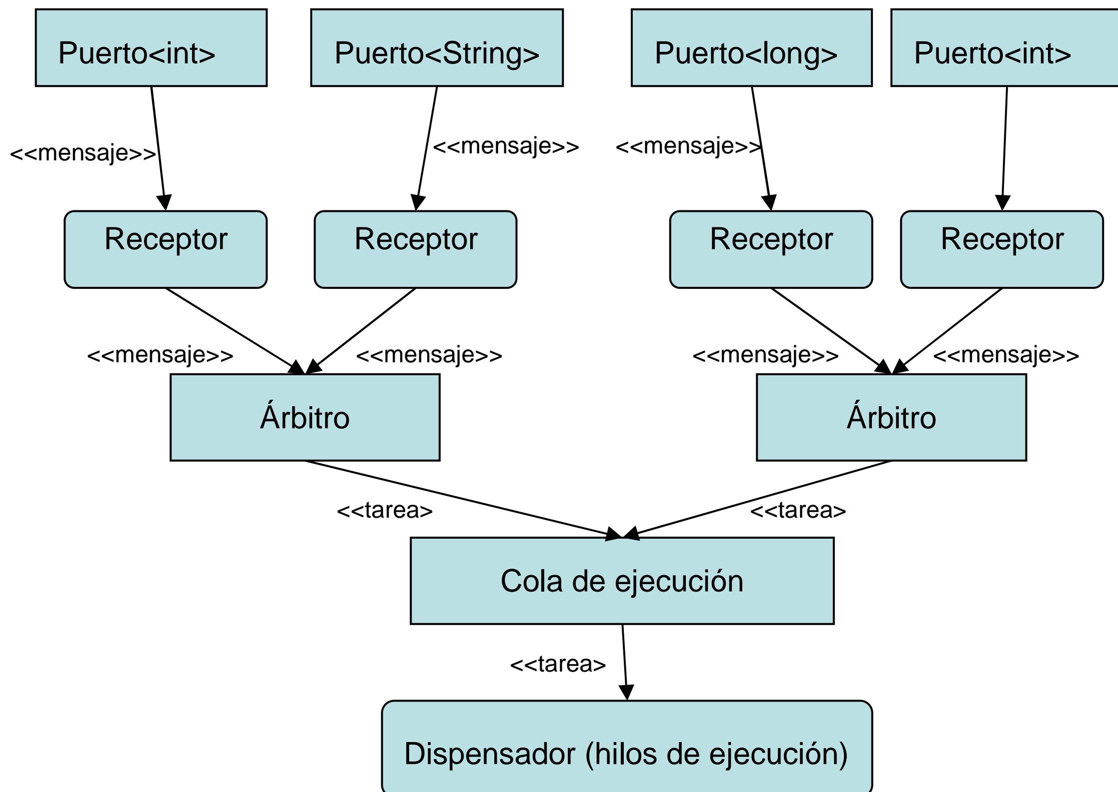

Figura 23.- Coordinación multitarea en CCR

Este esquema muestra la estructura de coordinación multitarea proporcionada por el CCR. Puede observarse que, en la parte superior, se encuentran distintos puertos de comunicación especializados en un determinado tipo de mensaje. Tras la llegada del mensaje adecuado a uno de los puertos, el receptor que observa a ese puerto informa al árbitro que lo creó sobre la recepción del mensaje. A continuación, este árbitro, en base al tipo de mensaje recibido, decide qué tarea (encapsulada como un objeto *ITask*) debe ejecutarse a continuación y la incluye en la cola de ejecución, en la que se encuentras las distintas tareas pendientes de ejecución. Finalmente, el Dispensador recorre la cola de ejecución y asigna hilos de ejecución a las distintas tareas en espera, cuando éstos se encuentran disponibles.

En el siguiente capítulo se describe una competición en la que, para participar, se requiere comprender y utilizar la herramienta MRDS.

Capítulo 4:
La competición Imagine Cup

En este capítulo se describen brevemente los orígenes de la competición Imagine Cup, así como las razones que fundamentan su existencia. Además, se realiza la descripción del problema planteado como desafío de la segunda ronda de la modalidad de Robótica y Algoritmos en su edición 2009.

4.1.- Historia y objetivos

La competición Imagine Cup[2] se celebró por primera vez en el año 2003 en Barcelona, siendo la presente su séptima edición. Fue fundada por Microsoft con el objetivo oficial de fomentar la participación juvenil en el desarrollo futuro de las nuevas tecnologías. Durante el transcurso de su historia, cada una de las ediciones ha versado sobre un tema diferente, siempre relacionado con situaciones y problemas de la vida real. En la presente edición el lema es "*Imagina un mundo en el que la tecnología ayuda a resolver los problemas más difíciles*". Este lema refleja la dinámica que esta competición ha presentado desde su creación, pues la temática siempre ha abogado por la imaginación de las nuevas generaciones con respecto al futuro. Este hecho se basa en la idea de que el mundo del mañana depende de los sueños de la juventud actual, por lo que intenta fomentar la inspiración y motivar la innovación, mediante la tecnología, entre los más jóvenes.

En su origen, sólo comprendía la categoría de Diseño Software, pero en la actualidad incluye ocho nuevas categorías: Desarrollo Embebido, Desarrollo de Juegos, Tecnologías de la información, Aplicaciones Web Híbridas, Fotografía, Cortometrajes, Diseño y Robótica y Algoritmos. En el presente proyecto, se elabora una solución preparada para participar en la segunda ronda de la categoría denominada "Robótica y Algoritmos". La primera ronda se celebró entre el 1 de marzo y el 1 de abril de 2009 y la segunda entre el 20 de abril y el 20 de mayo del mismo año. Por tanto, el tiempo total con el que se contó para la elaboración de la solución comprendió un mes exacto.

[2] Más información en http://imaginecup.com

La ronda final se celebra entre el 3 y el 7 de julio de 2009 en la ciudad de El Cairo, Egipto, y, para poder participar en ella, es preciso conseguir una de las seis primeras posiciones en la segunda ronda de la competición.

4.2.- Planteamiento del problema

La segunda ronda de la competición *Imagine Cup '09* se basa en el anteriormente existente desafío *Mars Rover Challenge* de la competición *RoboChamps*[3]. Este desafío requiere elaborar toda la lógica de control necesaria para que un robot del tipo NASA Mars Rover realice una misión de exploración sobre la superficie del planeta Marte.

En esta sección, se describen todas las características que es necesario conocer acerca del problema a resolver, así como una breve descripción de los acontecimientos reales que inspiraron el enunciado de dicho problema.

4.2.1.- Descripción general

El objetivo de la misión es realizar una exploración de reconocimiento y recolección de datos en los alrededores de la superficie del planeta Marte. En concreto, dicha exploración se realiza en torno al cráter denominado Endurance. Durante la misión, es necesario visitar una serie de puntos de interés seleccionados por los científicos. La localización exacta de estos puntos de interés es conocida y se encuentra definida por sus coordinadas cartesianas. En las proximidades de cada uno de dichos puntos, existen muestras de roca que deberán ser analizadas por nuestro robot utilizando un espectrómetro que, como veremos más adelante, se encuentra instalado en el brazo mecánico que éste posee.

Finalmente, tras obtener los resultados del análisis de las rocas situadas en cada uno de los puntos de interés, debe localizarse el escudo térmico que sirvió de protección al robot durante su llegada a la superficie del planeta. Las razones de esta búsqueda radican en que el conocimiento de los daños sufridos por dicho elemento de protección resulta muy útil para los científicos que lo diseñaron en futuros desarrollos. En lo que a esta búsqueda respecta, no se conoce la localización exacta del escudo térmico, sino que se encuentra en las proximidades de las últimas zonas de interés, por lo que deberá realizarse un rastreo de una amplia área.

---

[3] www.robochamps.com

4.2.2.- Robot NASA Mars Rover

El robot del que se dispone para la misión se denomina NASA Mars Rover. Este robot está compuesto por una serie de sensores y actuadores que se describen a continuación.

4.2.2.1.- Tracción

Se dispone de seis ruedas, dispuestas en dos hileras de tres ruedas a cada lado del robot, controladas por un dispositivo diferencial. Gracias a éste, es posible proporcionar distinta velocidad a cada uno de los segmentos de tres ruedas.

Debido a la implementación del servicio MRDS que lo simula, para controlar la velocidad de rotación y dirección de desplazamiento del robot, es necesario indicar la potencia que debe transmitirse a cada uno de los bloques de ruedas situados en los laterales izquierdo y derecho. Dicha potencia se encuentra definida en el rango [-1, 1], indicando los valores positivos movimiento de avance, mientras que los negativos indican movimiento de retroceso. Sin embargo, habrá de tenerse en cuenta que el desplazamiento del robot se realiza sobre terreno irregular y plagado, a su vez, de obstáculos, que provocarán el deslizamiento de las ruedas del robot, por lo que la odometría no será perfecta. Así pues, resulta necesario utilizar cierto tipo de sensores para conseguir el desplazamiento deseado.

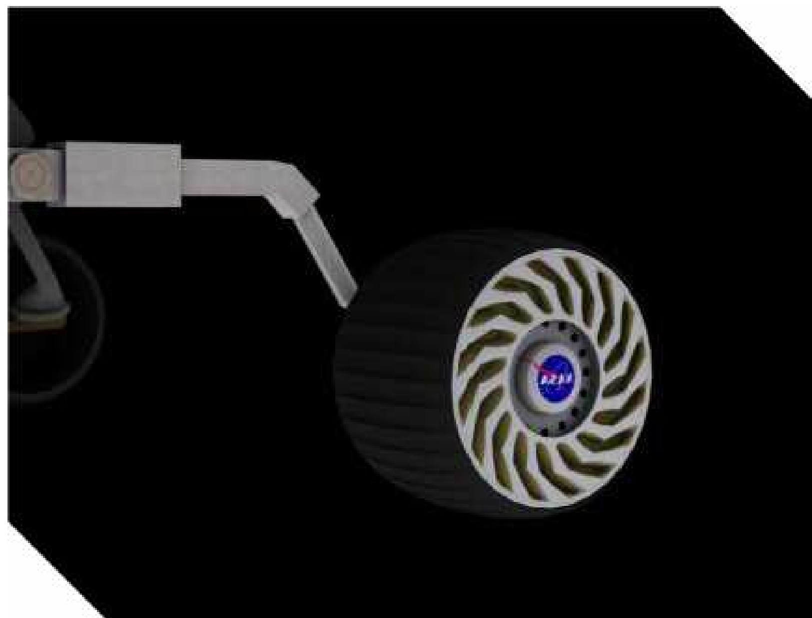

Figura 24.- Rueda del robot

4.2.2.2.- Cabeza articulada

Sobre el cuerpo del robot se encuentra un mástil terminado en una cabeza móvil. Esta cabeza puede rotar horizontal y verticalmente, pues en ella se encuentran localizadas la cámara panorámica y la cámara de navegación. Dichos movimientos se controlan a través del servicio MRDS denominado "*Pan-Tilt*", mediante el cual es posible indicar la realización de las rotaciones horizontales (Pan) y verticales (Tilt).

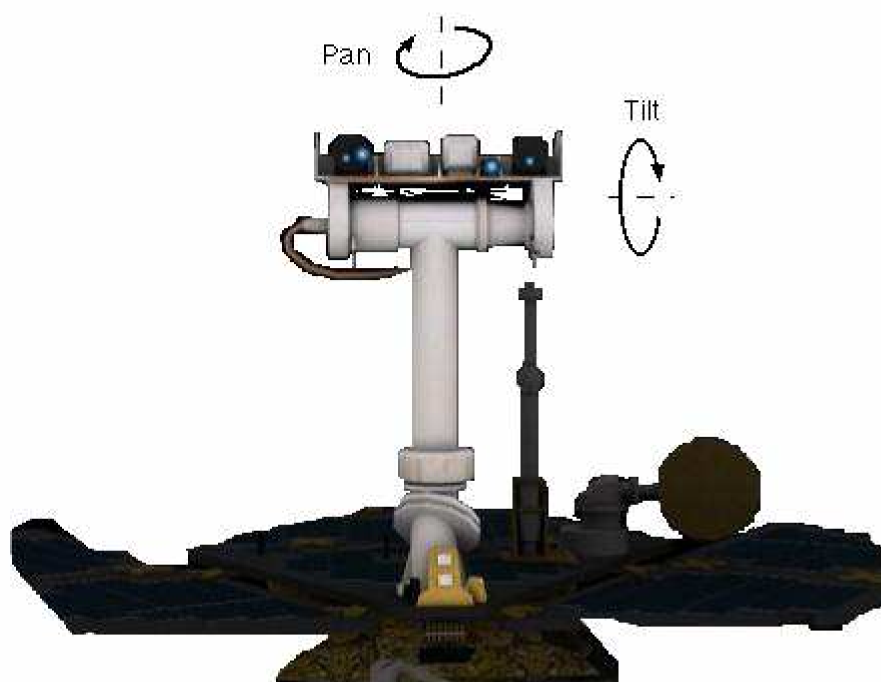

Figura 25.- Mástil y cabeza del robot

4.2.2.3.- Brazo articulado

El robot Rover posee un brazo mecánico de tres grados de libertad que imitan a las tres articulaciones que conforman un brazo humano: una muñeca, un hombro y un codo. Estas articulaciones se controlan a través del servicio MRDS denominado *ArticulatedArm*. Para ello, es necesario indicar el ángulo de rotación final en el que se desea situar cada una de las articulaciones del brazo, proporcionando valores de rotación para cada uno de sus cinco motores.

Las distintas articulaciones no permiten realizar movimientos de rotación completa, sino que están limitadas. Dichas limitaciones se detallan a continuación:

- **Muñeca:** está compuesta por dos motores que permiten realizar movimientos verticales de 340º y horizontales de 350º.

- **Codo:** sólo se precisa un único motor para controlarlo y se pueden realizar movimientos a lo largo de 290º.
- **Hombro:** en este caso, se precisan dos motores para realizar los distintos movimientos de la articulación. El motor que realiza las rotaciones verticales posee 70º de rotación, mientras que el encargado de realizar las rotaciones horizontales puede hacerlo a lo largo de 160º.

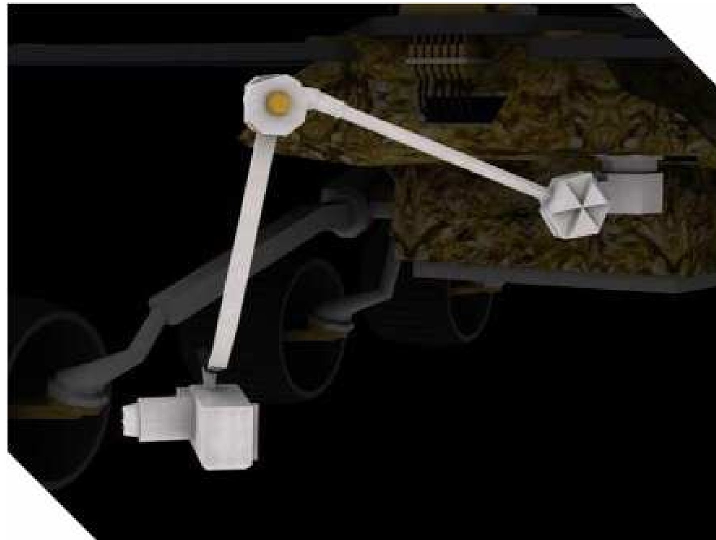

Figura 26.- Brazo mecánico del robot

4.2.2.4.- Cámaras

El robot posee cuatro cámaras que le permiten obtener información visual de su entorno. Dichas cámaras son de distinto tipo y se encuentran localizadas en diversas partes de la estructura del robot:

- **Cámara panorámica:** localizada en la cabeza del robot, captura imágenes en color con una resolución de 128x128 píxeles. Su campo de visión es de 16º, por lo que permite obtener imágenes de objetos alejados del robot.

- **Cámara de navegación:** de manera similar a la anterior, se encuentra en la cabeza del robot y obtiene imágenes a color con una resolución de 128x128 píxeles. Sin embargo, su campo de visión es de 45º, lo que permite obtener una imagen más amplia de las proximidades del robot.

- **Cámara frontal:** situada en la parte inferior delantera del cuerpo del robot, obtiene imágenes en blanco y negro con una resolución de 128x128 píxeles. Su campo de visión es de 120º, por lo que no permite obtener imágenes de elementos lejanos al robot. Sin embargo, percibe una gran área próxima a la parte delantera del robot, por lo que debe ser usada con fines de prevención de daños por colisión.

- **Cámara trasera:** similar en sus características a la cámara frontal, salvo porque se encuentra situada en la parte inferior trasera del robot. Por tanto, permite obtener imágenes de los elementos situados en las proximidades traseras del robot para evitar colisiones cuando el robot realice un movimiento de retroceso.

Para obtener los resultados del análisis de las imágenes obtenidas por cada una de las cámaras, se utiliza el servicio MRDS denominado *ImageProcessing*.

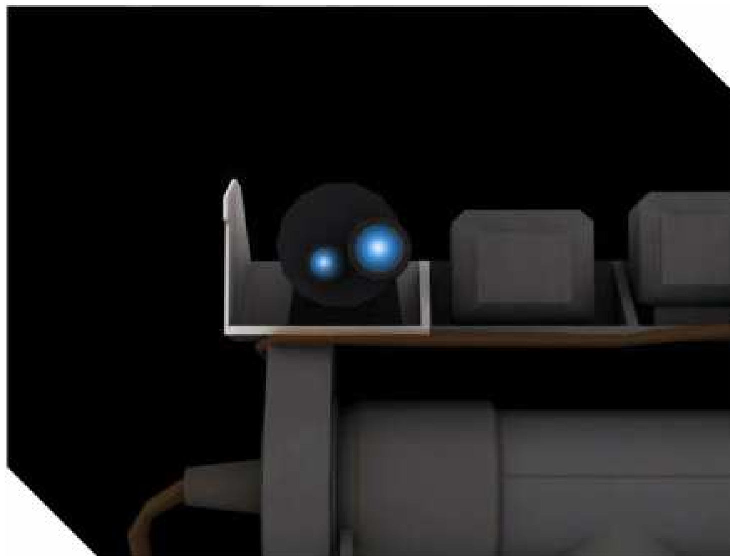

Figura 27.- Cámara del robot

4.2.2.5.- Espectrómetro

Este dispositivo se encuentra situado en el extremo final del brazo robótico. La finalidad de dicha herramienta es analizar la composición química de las muestras de roca localizadas en los lugares de interés. Para que dicho análisis tenga lugar, el dispositivo y la muestra han de entrar en contacto.

Como comentaremos más adelante, el servicio *MarsSpectrometer* está encargado de detectar si el espectrómetro ha entrado en contacto con una muestra y, en caso afirmativo, de obtener los valores que representan la composición química de la roca.

Figura 28.- Espectrómetro del robot

4.2.2.6.- Puerto de comunicaciones

Sobre la estructura que conforma el cuerpo del robot, se encuentra localizado un puerto de comunicaciones que permite realizar varias tareas:

- Conocer la localización exacta del robot a través de las coordenadas cartesianas que indican la posición del mismo.

- Conocer la orientación del robot a través de los ángulos de Tait-Bryan, que son un tipo específico de ángulos eulerianos, o coordenadas *Yaw, Pitch* y *Roll*.

- Obtener la denominación de la zona en la que se encuentra el robot, en el caso de que ésta sea conocida.

- Adquirir la lista de posiciones de interés que deberán ser visitadas en busca de muestras para análisis.

- Enviar a la Tierra un mensaje indicando la finalización de la misión.

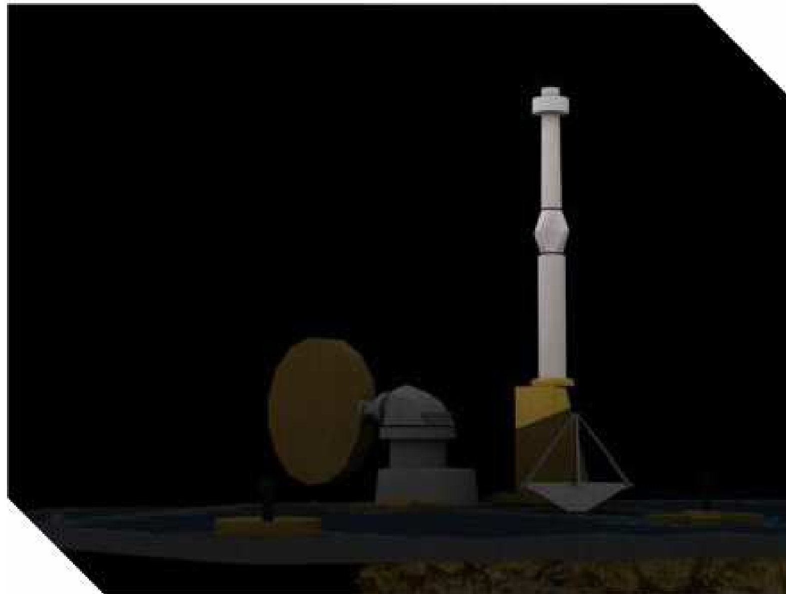

Figura 29.- Puerto de comunicaciones del robot

4.3.- Sistema de puntuación

Para poder evaluar el éxito de la misión, se encuentra instaurado un sistema de obtención de puntos. Mediante este sistema, una determinada cantidad de puntos se obtiene al llevar a cabo ciertas tareas.

El mencionado sistema, incluyendo las tareas a realizar y el número de puntos obtenido, es el siguiente:

- **Visitar un área de interés en el que no se ha estado con anterioridad**: 50 puntos.

- **Analizar una muestra de roca en un área en el que no se ha realizado ningún análisis anterior**: 50 puntos.

- **Localizar y visitar el escudo térmico**: 50 puntos.

- **Visitar las seis áreas de interés en orden, visitar el área en el que se encuentra el escudo térmico y, a continuación, enviar un mensaje de fin de misión a la Tierra**: 75 puntos.

Así pues, el número máximo de puntos que es posible obtener durante la misión es de 725, pues existen seis áreas de interés que pueden ser visitadas (6x5 = 300 puntos) y en las que es posible analizar muestras de roca (6x5 = 300 puntos), a las que se suman el escudo térmico (50 puntos) y la bonificación por realizar dichas tareas en orden (75 puntos).

4.4.- Contexto real

Tanto el escenario y elementos que conforman la misión, como los objetivos de la misma, se basan en una misión real de la NASA, en concreto del departamento denominado *Jet Propulsion Laboratory* (JPL), denominada *Mars Exploration Rover* (MER). En concreto, la misión se centra en el período comprendido entre el 30 de abril de 2004 y el 14 de enero de 2005, en el que el robot *Opportunity* realizó una exploración del cráter marciano *Endurance*, así como examinó el escudo térmico que lo protegió durante su aterrizaje en Marte.

En esta sección se describen brevemente ambos acontecimientos.

4.4.1.- La misión Mars Exploration Rover

Esta misión comenzó con el lanzamiento de los robots gemelos MER-A, actualmente denominado *Spirit*, y MER-B, rebautizado como *Opportunity*, los días 10 de junio y 3 de julio de 2003, respectivamente, con el propósito de obtener respuestas acerca de la existencia de agua en Marte. Ambos robots son del tipo *Mars Exploration Rover* y pueden llevar a cabo tareas similares a las realizadas por un geólogo humano. *Spirit* y *Opportunity* llegaron a la superficie de Marte los días 3 y 25 de enero de 2004, respectivamente, tras seis meses de viaje espacial. Entre los distintos objetivos de la misión se encuentran, principalmente, la búsqueda y clasificación de un amplio conjunto de rocas y terrenos que poseen pistas sobre la actividad del agua en Marte en el pasado. Por ello, los robots tomaron tierra en distantes localizaciones del planeta Marte, en las que se evidenciaba una pasada actividad acuática.

Spirit aterrizó en el cráter de *Gusev*, cuenca de un posible lago aparecido en el cráter formado por un gran impacto sobre la superficie. Por otro lado, *Opportunity* tomó tierra en la llanura denominada *Meridiani Planum*, próxima al ecuador marciano y en la que depósitos de hematita evidencian signos de un pasado acuoso.

Figura 30.- Fotografía del Meridiani Planum

Tras el aterrizaje, cada una de las naves se abrió para permitir que los rovers se desplegasen y tomasen imágenes panorámicas de los alrededores. Dichas imágenes proporcionaron a los científicos de la Tierra elegir qué localizaciones debían ser visitadas debido a su gran interés geológico en lo que a una posible pasada actividad acuática respecta. Inmediatamente después, los rovers se desplazaron a cada uno de estos puntos de interés para realizar estudios científicos. Para ello, los rovers estaban dotados de los siguientes instrumentos científicos:

- **Cámara Panorámica** (Pancam): para determinar la mineralogía, textura y estructura del terreno local.

- **Mini espectrómetro de emisión térmica** (Mini-TES): para identificar rocas y terrenos prometedores en los que realizar detenidos análisis en los que se pudiesen determinar las condiciones en las que se formaron las rocas marcianas. Este instrumento está diseñado para observar el cielo marciano con la finalidad de obtener los perfiles de temperatura que caracterizan a la atmósfera de Marte.

- **Espectrómetro de Mössbauer** (MB): para realizar análisis en primer plano de la mineralogía de los terrenos y rocas ricos en hierro.

- **Espectrómetro de rayos X y partículas alfa** (APXS): para realizar análisis en primer plano de las cantidades de elementos que componen las rocas marcianas.

- **Imanes**: para recolectar partículas magnéticas de polvo. Dichas partículas son analizadas por el espectrómetro de Mössbauer y el Espectrómetro de rayos X y partículas alfa para determinar la proporción de partículas magnetizadas con respecto a las no magnetizadas.

- **Cámara microscópica** (MI): para obtener imágenes de alta resolución de las rocas a escala microscópica.

- **Herramienta de abrasión** (RAT): para pulir rocas con el fin de eliminar polvo y material deteriorado de la superficie de las rocas, exponiendo así el material idóneo para realizar el análisis de la composición de las mismas.

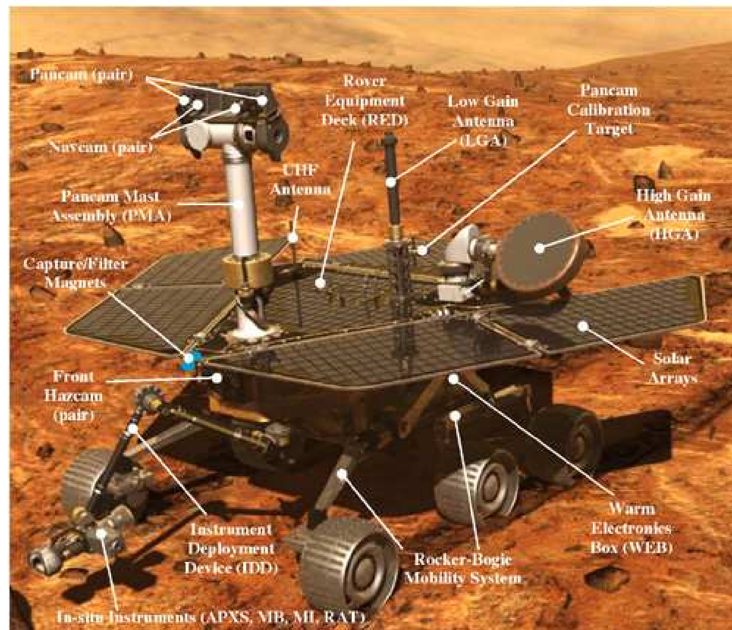

Figura 31.- Representación de un robot Mars Rover

En un principio, las expectativas de vida de los robots fueron de 90 sols. Sin embargo, en la actualidad ambos rovers continúan su recorrido y análisis de la superficie marciana. En su desplazamiento entre localizaciones, los robots realizan las mismas investigaciones geológicas que un ser humano realizaría si se encontrase en la misma localización. Para ello, cuentan con una serie de cámaras montadas en un mástil que se encuentra a una altura de 1,5 metros sobre el suelo, con las que es posible obtener imágenes estereoscópicas alrededor de 360º, de manera similar a las obtenidas por un ser humano. Por otro lado, los rovers cuentan con un brazo mecánico que puede realizar movimientos similares a los realizados por un brazo humano, incluyendo las articulaciones del codo y la muñeca, permitiendo así colocar los instrumentos de análisis en contacto directo con los elementos a analizar. Para ello, en el extremo en el que debería localizarse la mano, el brazo robótico contiene la cámara microscópica y la herramienta de abrasión, además de los distintos espectrómetros con los que se llevarán a cabo los diferentes análisis.

Entre los distintos descubrimientos que se realizaron y se siguen realizando, pues en el momento de redacción del presente documento la misión se encuentra en proceso, se pueden destacar los siguientes:

- Se han encontrado evidencias de la pasada existencia de agua en Marte, en la llanura denominada *Meridiani Planum*. De hecho, los resultados apuntan a la existencia de un mar salado.

- Se ha descubierto el primer meteorito en un cuerpo celeste distinto a la Tierra.

Puede consultarse un diario del transcurso de la misión en la página oficial de la NASA:

http://marsrovers.jpl.nasa.gov/mission/wir/

4.4.2.- Exploración del cráter Endurance y el escudo térmico

El 30 de abril de 2004, tres meses después de su llegada a Marte y tras visitar el cráter *Eagle*, el rover denominado *Opportunity* se trasladó hasta las proximidades del cráter denominado *Endurance*. El estudio de este cráter resultaba de gran interés científico, pues en él se podían observar distintas capas de roca.Durante el mes de mayo, *Opportunity* rodeó el cráter realizando análisis de las rocas que se encontraban en esta localización. El 4 de junio de 2004 se decidió realizar una incursión al interior del cráter y, el día 8 de ese mismo mes, *Opportunity* se introdujo en el cráter. Ese mismo día volvió a abandonar el cráter, pues el objetivo era identificar si se podría salir del mismo.

Figura 32.- Panorámica del cráter Endurance realizada por el robot Opportunity

Tras comprobar que era posible abandonar el cráter, el rover se introdujo en lo más profundo de la abertura el 12 de junio. Durante seis meses se realizaron estudios en distintas partes de *Endurance*, siendo las más remarcables el afloramiento rocoso denominado *Karatepe* y los precipicios de Burns (*Burns Cliffs*). Posteriormente, en diciembre de 2004, *Opportunity* abandonó el cráter y se encaminó hacia los restos de su escudo térmico para observar los daños que éste había sufrido. Durante este trayecto, descubrió el primer meteorito hallado en un cuerpo celeste distinto a la Tierra, bautizado con el nombre de Roca del Escudo Térmico (*Heat Shield Rock*).

Figura 33.- Roca del Escudo Térmico

Finalmente, en enero de 2005, se alcanzó el escudo térmico y se realizaron análisis del mismo durante 25 días marcianos, abandonando el área en dirección al cráter Argo.

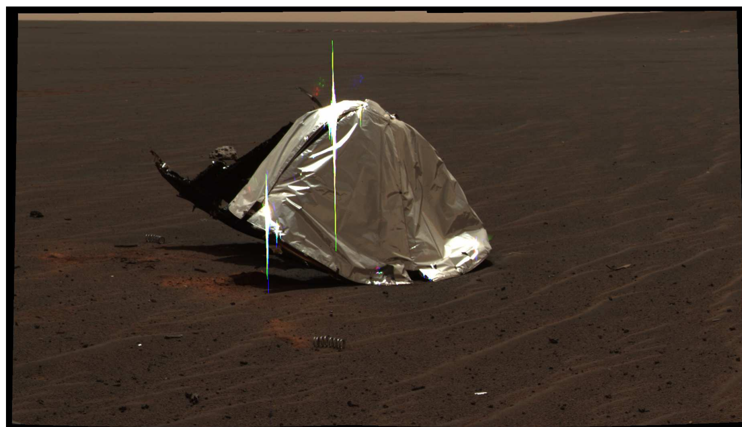

Figura 34.- Escudo térmico en el que viajó el robot Opportunity

Capítulo 5:
Código de Partida

En este capítulo se realiza una descripción del código inicial que se obtiene al realizar la instalación de la aplicación. Además, se ejemplifica con ejemplos gráficos y se enfatiza la utilidad de cada uno de los elementos de un servicio con la finalidad de aportar ejemplos prácticos a toda la teoría incluida en el capítulo 3.

Tras realizar la instalación del desafío, se han generado una serie de ficheros Visual Studio que contienen la estructura y funcionalidad básicas del robot. Estos ficheros son los siguientes:

- **ImageProcessingTypes.cs**: contiene la especificación del ámbito o *namespace* del servicio *ImageProcessing*, encargado de procesar las imágenes adquiridas por las distintas cámaras del robot. Por tanto, se definen los distintos tipos de datos que serán manejados por el servicio. Estos tipos de datos serán intercambiados en mensajes y accedidos para obtener distintos tipos de información.

- **ImageProcessing.cs**: implementa la funcionalidad del servicio encargado de procesar las diferentes imágenes obtenidas a través de las cámaras del robot. Además, contiene la lógica necesaria para encapsular en un mensaje y enviar al servicio que controla el robot, los resultados de los distintos análisis realizados a los fotogramas adquiridos.

- **MarsChallengerTypes.cs**: en éste se especifica el ámbito del servicio *MarsChallenger*, que es el que implementa el comportamiento principal del robot. De manera similar a la función del primer fichero, en éste se definen los tipos de datos que serán intercambiados y manejados por el servicio al que especifica.

- **MarsChallenger.cs**: contiene el código fuente con toda la lógica que controlará al robot. En él se describen las distintas operaciones llevadas a cabo por el servicio, así como información de sus servicios asociados y su interacción con los mismos. Por otro lado, también se define el servicio que controlará al robot en su totalidad, incluyendo descripciones de sus distintos componentes.

En estos ficheros iniciales se aporta la estructura principal de la aplicación, pero sin ninguna funcionalidad implementada. En éstos deberá implementarse toda la lógica de control necesaria para superar los retos de navegación anteriormente descritos. Con la finalidad de proporcionar una visión más clara de los servicios iniciales, se incluye una descripción gráfica de la orquestación de los distintos servicios que conforman la aplicación.

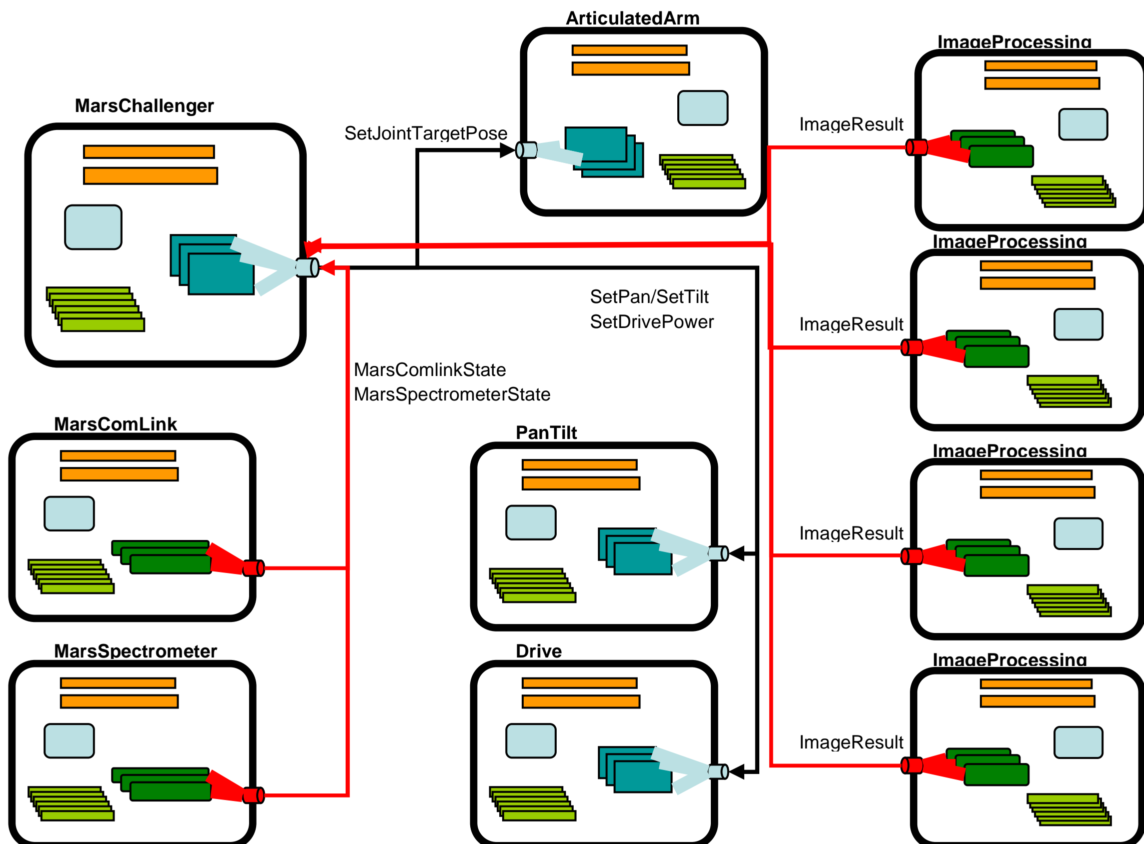

Figura 35.- Orquestación de los servicios

En este caso, se muestran los distintos servicios asociados al servicio *MarsChallenger* y los tipos de mensajes entre ellos. Se muestran, con flechas negras, las distintas solicitudes que se pueden realizar entre los puertos principales y, con flechas rojas, los mensajes de notificación entre servicios.

Además, en los anteriores ficheros se describe la funcionalidad completada aporta por los dos servicios que deberemos completar, el servicio *ImageProcessing* y el servicio *MarsChallenger*. A continuación se incluye una descripción más detallada de los componentes que conforman estos dos servicios con la finalidad de ilustrar con un ejemplo práctico la información anteriormente expuesta sobre la programación robótica en MRDS.

5.1.- Servicio ImageProcessing

Este se encarga de recibir y procesar las imágenes obtenidas a través de los servicios de cámara. Una vez procesadas las distintas imágenes, los resultados del análisis son enviados a través del puerto de suscripción a los diferentes servicios suscritos a éste. En la siguiente figura se muestran la estructura y componentes de este servicio:

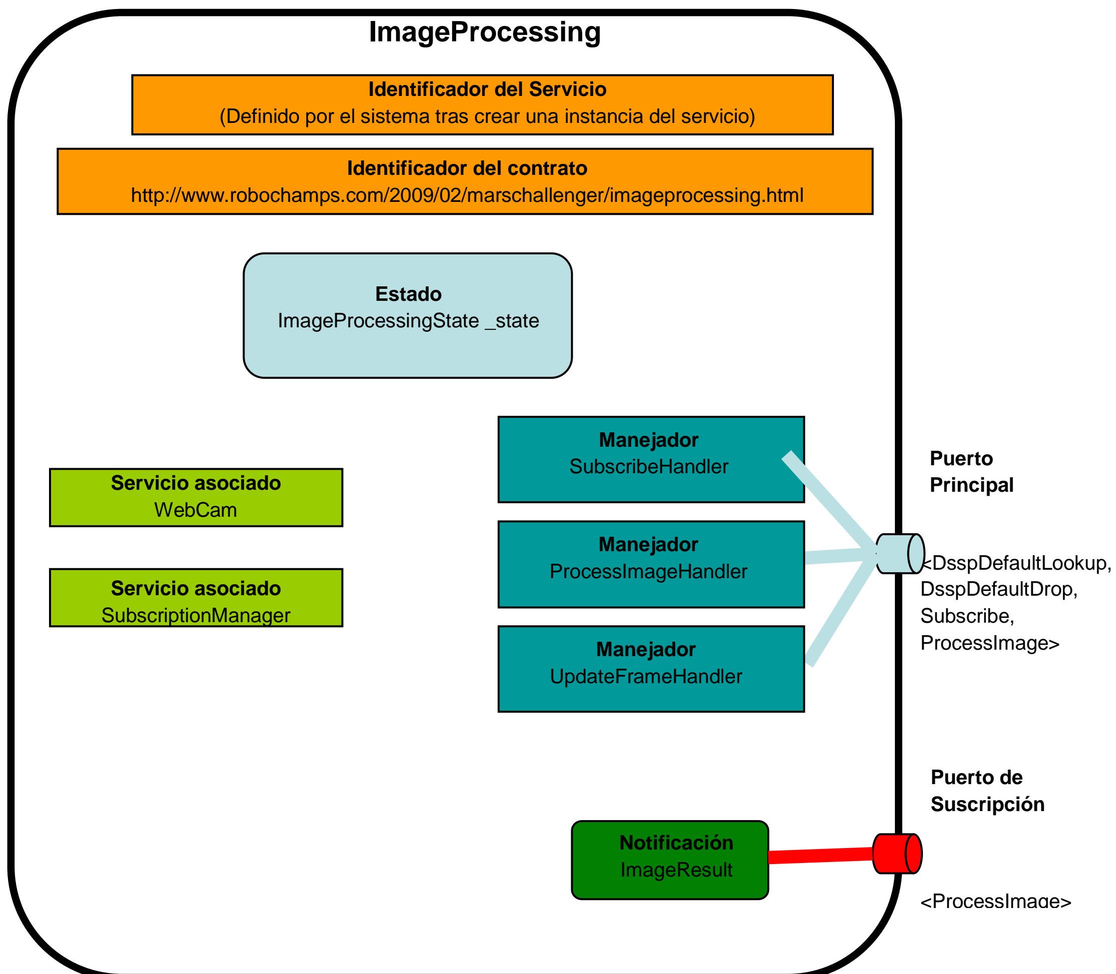

Figura 36.- Servicio ImageProcessing

Los ficheros en los que se puede encontrar la definición de cada uno de estos componentes se describen en los siguientes apartados.

5.1.1.- Fichero ImageProcessingTypes

Las primeras líneas de código que se encuentran en este documento, indican a qué librerías se hará referencia, por lo que éstas deberán ser importadas durante la compilación y posteriormente montadas durante la ejecución, al tratarse de librerías dinámicas.

```
using System;
using Microsoft.Ccr.Core;
using System.Collections.Generic;
using System.ComponentModel;
using Microsoft.Dss.ServiceModel.DsspServiceBase;
using Microsoft.Dss.Core.Attributes;
using Microsoft.Dss.ServiceModel.Dssp;
using W3C.Soap;
using submgr = Microsoft.Dss.Services.SubscriptionManager;
using webcam = Microsoft.Robotics.Services.WebCam.Proxy;
```

Puede comprobarse que la directiva "using" es utilizada para indicar las distintas librerías. Este procedimiento es similar al realizado por las directivas "import" en Java o "include" en C. A continuación, comienza la descripción del ámbito (namespace) del servicio al que define, denominado *Microsoft.Robotics.Robochamps.MarsChallenger.Imageprocessing.*

El primer elemento que se incluye es la declaración de la clase *Contract,* que contiene el atributo que representa al identificador de contrato del servicio. Se trata de una clase sellada, es decir, que no admite herencia a partir de ella, debido al modificador *sealed*, similar a la directiva *final* de Java. El único atributo que esta clase contiene es la constante *Identifier* que se corresponde con el anteriormente mencionado identificador de contrato:

```
    public sealed class Contract
    {
        [DataMember]
        public const String Identifier =
"http://www.robochamps.com/2009/02/marschallenger/imageprocessing.html
";
    }
```

Podemos comprobar que la constante *Identifier* se encuentra precedida del atributo DataMember que, como explicaremos más adelante, registra esta variable como serializable. Inmediatamente después, se incluye una serie de elementos, precedidos por el atributo *DataContract*. Cada uno de estos elementos define un tipo de datos que será manejado por el servicio. Sin embargo, para que éstos puedan ser reconocidos e intercambiados por otros servicios asociados al nuestro, debe incluirse el atributo *DataContract* para indicar que se desea poder serializar un determinado elemento mediante un serializador.

El trabajo de un serializador consiste en convertir los datos que representan a un elemento de un determinado ámbito en un fichero XML y viceversa. La necesidad de estos serializadores radica en el hecho de que la información intercambiada a través de los puertos se realiza a través de ficheros XML. Por tanto, cualquier información que pueda intercambiarse a través de los puertos debe ser serializable, es decir, debe existir un serializador para la misma.

A la hora de crear una clase serializable, debemos tener en cuenta que resulta necesario indicar qué componentes de la misma deben ser serializados en el fichero XML. Para ello, se utiliza el atributo *DataMember*. Gracias a éste, podemos definir qué componentes de la clase serán serializables y, por tanto, podrán intercambiarse a través de los puertos.

Por tanto, podemos comprobar que existen dos clases serializables:

- **ImageProcessingState**: esta clase representa el estado del servicio *ImageProcessing*. Contiene una única variable de tipo temporal, *LastProcessing*, que indica el momento en el que se recibió el último fotograma desde los servicios de cámara web.

- **ImageResult**: clase que encapsula los resultados obtenidos tras realizar el análisis de un fotograma obtenido a través de los servicios de cámara. En este momento, sólo aporta información sobre el momento en el que se recibió la última imagen.

El siguiente elemento del ámbito que se describe es el puerto principal del servicio. Para ello, es necesario incluir el atributo *ServicePort* para que el siguiente puerto declarado se registre como el puerto principal del servicio:

```
    [ServicePort]
    public class ImageProcessingOperations :
PortSet<DsspDefaultLookup, DsspDefaultDrop, Subscribe, ProcessImage>
    {
      […]
    }
```

Como se puede comprobar, el puerto principal es una clase denominada *ImageProcessingOperations* que se comporta de la misma manera que una instancia de la clase *Portset*, descrita en el apartado 3.2.2.1. Por otro lado, se puede observar que los tipos de peticiones que se pueden realizar a través de este puerto son *DsspDefaultLookup, DsspDefaultDrop*, *Subscribe* y *ProcessImage*.

Las dos primeras peticiones corresponden a clases que están predefinidas, cuyo cometido es informar sobre las operaciones que soporta un servicio, en el caso de la primera, y solicitar la parada del servicio, en el caso de la segunda. La tercera clase indica que es posible solicitar una suscripción al servicio y la cuarta, como se explica a continuación, permiten actualizar el estado del servicio. Posteriormente, se encuentra la definición de las clases *Subscribe* y *ProcessImage*. Estas clases encapsulan mensajes que representan a dos de las operaciones permitidas por el servicio, pues están incluidas en la declaración del puerto principal del mismo. A continuación, se describe brevemente las implicaciones que derivan cada una de las definiciones.

En la definición de la clase Subscribe, se indica que deriva de una clase predefinida del mismo nombre. Los parámetros que toma son, en primer lugar, el cuerpo del mensaje correspondiente a la operación que encapsula, siendo éste del tipo *SubscribeResponseType*, y, en segundo lugar, el tipo de respuestas que serán emitidas por el servicio al recibir la petición. Estas respuestas serán de dos tipos:

- **SubscribeResponseType**: tipo de respuesta ya predefinida en la clase que hereda y que, como su nombre indica, contiene información referente a la respuesta a una petición de suscripción.

- **Fault**: si ha ocurrido algún error y el servicio no puede emitir la respuesta adecuada.

Toda esta información se encuentra implementada en las siguientes líneas de código:

```
    public class Subscribe : Subscribe<SubscribeRequestType,
PortSet<SubscribeResponseType, Fault>>
      {
        public Subscribe() : base() { }
        public Subscribe(SubscribeRequestType body) : base(body) { }
        public Subscribe(SubscribeRequestType body,
PortSet<SubscribeResponseType, Fault> responsePort) : base(body,
responsePort) { }
       }
```

Como podemos observar, todos los constructores de la clase Subscribe son los mismos que los de la clase homónima de la que hereda, hecho que se indica explícitamente mediante el uso de dos puntos. Por otro lado, también se define la clase ProcessImage. De manera similar a la clase anterior, esta clase hereda el comportamiento de la clase predefinida denominada Update. Los tipos de parámetros que puede recibir en su instanciación son: una instancia de la clase ImageResult, que conforma el cuerpo principal del mensaje, y los dos tipos de respuestas que pueden ser emitidas al realizar una petición ProcessImage:

- **DefaultUpdateResponseType**: respuesta emitida tras recibir la solicitud de actualización del servicio.

- **Fault**: emitida si ha ocurrido algún error durante el proceso de la solicitud.

La comprensión de estas dos declaraciones resulta de gran utilidad para el programador novel, pues permite comprobar cómo se definen las operaciones que soporta un servicio, así como qué tipo de información debe incluirse en las peticiones y qué tipos de contestaciones pueden obtenerse tras realizarlas. En este caso, las clases que encapsulan el cuerpo del mensaje de petición, *SubscribeResponseType* y *DefaultUpdateResponseType*, se encuentran predefinidas, pero podría no ser así y tratarse de clases creadas por el programador con la serie de atributos que éste considere necesaria.

5.1.2.- Fichero ImageProcessing.cs

En este fichero se incluyen todas las directivas necesarias para realizar un procesado de las imágenes obtenidas a través de los distintos servicios de cámara web. En primer lugar, se encuentran especificadas las distintas librerías dinámicas que deberán ser importadas. Inmediatamente a continuación, comienza la descripción del servicio. Se incluye el nombre a mostrar del servicio, mediante el atributo *DisplayName*, pero no debemos confundir este nombre con el del contrato que identifica al servicio. Además, se incluye una breve descripción de la funcionalidad realizada por el servicio, utilizando para ello el atributo *Description*.

A continuación, se especifica el identificador del contrato del servicio mediante la línea:

```
[Contract(Contract.Identifier)]
```

Como puede comprobarse, se utiliza como elemento identificador la variable *Identifier* de la clase *Contract*, cuya definición se encuentra en el apartado anterior. Tras esta declaración de atributos, se declara la clase *ImageProcessingService*, que deriva de la clase *DsspServiceBase*. En esta clase se define todo el comportamiento del servicio que se implementará. Al principio de esta clase, se definen los distintos componentes del servicio. Dichos componentes fueron descritos teóricamente en el capítulo 3.

5.1.2.1.- Estado del servicio

El estado del servicio se define de la siguiente manera:

```
[ServiceState]
private ImageProcessingState _state = new ImageProcessingState();
```

El atributo *ServiceState* permite registrar la variable *_state* como representante del estado del servicio. Este estado es una instancia de la clase *ImageProcessingState*, definida anteriormente, y, por tanto, cualquier tipo de información que deseemos que forme parte del estado del servicio debe ser incluida como atributo de las instancias de esa clase.

5.1.2.2.- Puerto Principal

El puerto principal se define de la siguiente manera:

```
[ServicePort("/ImageProcessingService", AllowMultipleInstances = true)]
private ImageProcessingOperations _mainPort = new ImageProcessingOperations();
```

Con la inclusión del atributo *ServicePort* se registra la instancia *_mainPort* como puerto principal del servicio. Este puerto es una instancia de la clase ImageProcessingOperations que, como se vio anteriormente, es un *PortSet* que permite recibir mensajes de los tipos *DsspDefaultLookup*, *DsspDefaultDrop*, *Subscribe* y *ProcessImage*. Al declararse el puerto principal de este tipo, se indica que estos cuatro tipos de mensajes son los que definen las cuatro operaciones que es posible solicitar al servicio.

Por otro lado, en el atributo *ServicePort* se indica que es posible poseer múltiples instancias de este puerto, mediante la directiva *AllowMultipleInstances = true*. De esta manera, a cada instancia del servicio se le asignará un nombre de servicio distinto, permitiendo diferenciar una instancia de las demás. Así pues, es posible que existan, simultáneamente, distintos servicios asociados que contengan una distinta instancia de este puerto y que puedan, por tanto, comunicarse con una instancia diferente del servicio.

5.1.2.3.- Servicios Asociados

El servicio *ImageProcessingService* se relaciona con una serie de servicios asociados, o *partners*, para proporcionar su funcionalidad. Estos servicios son los siguientes:

- **WebCam**: implementa la funcionalidad de una cámara web.
- **SubscriptionManager**: servicio del sistema encargado de manejar las suscripciones entre servicios.

Para que nuestro servicio se pueda comunicar con cada uno de estos servicios, es necesario poder acceder a sus puertos principales. Para ello, deben registrarse como servicios asociados de la siguiente manera:

```
        [Partner("Camera", Contract = webcam.Contract.Identifier,
CreationPolicy = PartnerCreationPolicy.UsePartnerListEntry)]
        private webcam.WebCamOperations _camPort = new
webcam.WebCamOperations();
        private webcam.WebCamOperations _camNotify = new
webcam.WebCamOperations();

        [SubscriptionManagerPartner]
        private   submgr.SubscriptionManagerPort   _submgrPort   =   new
submgr.SubscriptionManagerPort();
```

El atributo *Partner* toma como parámetros el nombre que se asigna al servicio asociado, el identificador del servicio asociado y la política de creación del mismo. Las políticas existentes permiten utilizar una instancia ya creada del servicio asociado o crear una nueva. En este caso, se usará una instancia ya creada del mismo. De esta manera, la instancia *_camPort* nos permite comunicarnos con el servicio que actúa como cámara web. A través de esta instancia se pueden solicitar las operaciones oportunas a dicho servicio. Además, se crea otra instancia de ese tipo de puerto, _camNotify, que no se registra como puerto principal, pues, como veremos más adelante, será utilizado como puerto de notificación.

Por otro lado, el atributo *SubscriptionManagerPartner* permite declarar una nueva asociación con el servicio *SubscriptionManager*, encargado de manejar las suscripciones entre servicios. Para obtener una mayor comprensión de esta etiqueta, podemos observar la sentencia original que era utilizada en versiones anteriores con el mismo propósito:

```
[Partner("SubMgr",      Contract      =      submgr.Contract.Identifier,
CreationPolicy = PartnerCreationPolicy.CreateAlways)]
```

5.1.2.4.- Manejadores

Este servicio contiene distintos manejadores que permiten procesar la información contenida en los mensajes que se reciben por los diferentes puertos, ya sea el puerto principal o los puertos de comunicación con los servicios asociados.

Los manejadores, o *handlers*, incluidos son los siguientes:

- **SubscribeHandler**: recibe y gestiona las solicitudes de suscripción por parte de otros servicios. Por tanto, se ejecuta cada vez que se recibe un mensaje del tipo *Subscribe* a través del puerto principal.

- **ProcessImageHandler**: se encarga de solicitar al servicio SubscriptionManager la notificación de este evento a los servicios suscritos al servicio ImageProcessingService.

- **UpdateFrameHandler**: se ejecuta siempre que se recibe un nuevo fotograma desde el servicio de cámara web. Este manejador modifica el estado del servicio con la información recibida, invoca al método procesador de imagen correspondiente y envía el resultado al puerto principal encapsulado en una instancia del tipo ProcessImage. Por tanto, el servicio, al recibir este mensaje de actualización a través de su puerto principal, envía notificaciones a los servicios suscritos a él.

5.1.2.5.- Lógica de ejecución

Además de los elementos que conforman un servicio DSS, se incluye una serie de funciones y atributos que conforman la lógica de control que dirigirá el flujo de ejecución de las distintas funcionalidades. En el caso del servicio ImageProcessingService, se utilizan una serie de atributos con distinta finalidad:

- **Dictionary _processors**: este atributo representa una instancia de la clase *Dictionary<String, IImageProcessor>*. Gracias a ésta, es posible asociar un tipo determinado de instanciación de la interfaz *IImageProcessor* a una variable de tipo *String*. De esta manera, y como veremos a continuación, se asociará un tipo de procesador de imagen distinto a cada instancia del servicio dependiendo del *String* que actúe como identificador del mismo.

- **IImageProcessor _processor**: procesador de imagen utilizado por el servicio. Dependerá del identificador asociado al servicio durante su instanciación.

- **ImageProcessingService**([…]): constructor del servicio. Inicializa el servicio y asocia los distintos procesadores de imagen con un nombre de servicio determinado.

- **Interface IImageProcessor:** interfaz que encapsula el resultado obtenido tras procesar un fotograma obtenido desde el servicio de cámara. Requiere que todas las clases que hereden de ella definan un método denominado *Process*, que tome por parámetro una imagen y devuelva una instancia del tipo *ImageResult*.

- **IImageProcessor pancamProcessor:** procesador de imagen encargado de analizar las imágenes obtenidas a través del servicio de cámara web que representa la cámara panorámica del robot.

- **IImageProcessor navcamProcessor:** procesador de imagen que analiza las imágenes de la cámara de navegación del robot, obtenidas a través de un servicio de cámara web.

- **IImageProcessor basicCamProcessor:** procesador de imagen encargado de analizar las imágenes correspondientes a las cámaras frontal y trasera.

- **Start()** : es el primer procedimiento que se ejecuta al inicializar el servicio. Todo servicio DSS debe contener este método.

El servicio *Start* se define de la siguiente manera:

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

// Gets the processor associated to this service
_processors.TryGetValue(ServiceInfo.Service, out _processor);

// Initialization
_state.LastProcessing = DateTime.MinValue;

    // Subscribe to webcam notification
    MainPortInterleave.CombineWith(Arbiter.Interleave(
        new TeardownReceiverGroup(),
        new ExclusiveReceiverGroup
        (

    Arbiter.ReceiveWithIterator<webcam.UpdateFrame>(true,
_camNotify, UpdateFrameHandler)
        ),
    new ConcurrentReceiverGroup()));
    camPort.Subscribe(_camNotify, typeof(webcam.UpdateFrame));
}
```

En este caso, se inicia el servicio invocando el método *Start* de la clase *DsspServiceBase*. A continuación, se obtiene el procesador de imagen correspondiente al nombre del servicio y se inicializa la variable *LastProcessing*, descrita con anterioridad y perteneciente al estado del servicio.

Finalmente, se realiza la suscripción al sistema de notificación y el registro del manejador *UpdateFrameHandler*. Para ello, se accede al árbitro *MainPortInterleave*. Este árbitro de tipo *Interleave* es creado por defecto en la clase *DsspServiceBase* y en él deben registrarse los distintos receptores del servicio. Como podemos observar, este árbitro clasifica los árbitros anidados que contiene en tres categorías:

- **TearDownReceiverGroup**: estos árbitros se ejecutan con preferencia sobre todos los demás y, tras ejecutarse, se desactiva el árbitro *InterLeave* independientemente de si existen otros receptores en ejecución.

- **ExclusiveReceiverGroup**: los árbitros incluidos en este grupo no pueden ejecutarse al mismo tiempo. Si uno de ellos se está ejecutando, el resto deberá esperar a que éste termine, aunque se haya recibido un mensaje en el puerto que monitorizan.

- **ConcurrentReceiverGroup**: estos receptores se pueden ejecutar concurrentemente, siempre y cuando exista el número adecuado de hilos de ejecución disponibles. Sin embargo, no se ejecutarán mientras un receptor de cualquiera de los otros grupos se esté ejecutando.

Debido a que pertenece al segundo grupo, el manejador *UpdateFrameHandler* se ejecutará exclusivamente. Además, se utiliza un árbitro del tipo *ReceiveWithIterator*. Este árbitro utiliza un iterador para ejecutar el procedimiento manejador. El receptor asociado escucha el puerto *_camNotify*, a través del cual llegarán las notificaciones por parte del servicio de cámara web. Cuando éstas notificaciones sean recibidas, el mensaje es transferido al manejador y éste ejecuta las acciones correspondientes. Este receptor estará siembre habilitado, hasta que sea detenido explícitamente, pues el primer parámetro, *true,* indica la persistencia del receptor. En caso contrario, sólo se mantendría disponible para recibir un único mensaje y, después, se deshabilitaría automáticamente.

Por otro lado, en la suscripción al servicio de cámara web, debe resaltarse que se utiliza el puerto principal de éste servicio, denominado *_camPort*, para solicitar dicha suscripción, pero se indica un puerto diferente, *_camNotify*, para recibir las actualizaciones. Así pues, resulta notable que dichos puertos son distintos, siendo el primero una instancia del puerto principal del servicio de cámara y el segundo un puerto de suscripción.

5.2.- Servicio MarsChallenger

Este servicio incluye la lógica de control que determina el comportamiento del robot. A él llega la información que se obtiene a través de los distintos sensores y desde él se pueden controlar el movimiento del robot y sus dispositivos instalados, como el brazo robótico y la cabeza. A continuación se incluye una representación gráfica de los componentes de este servicio:

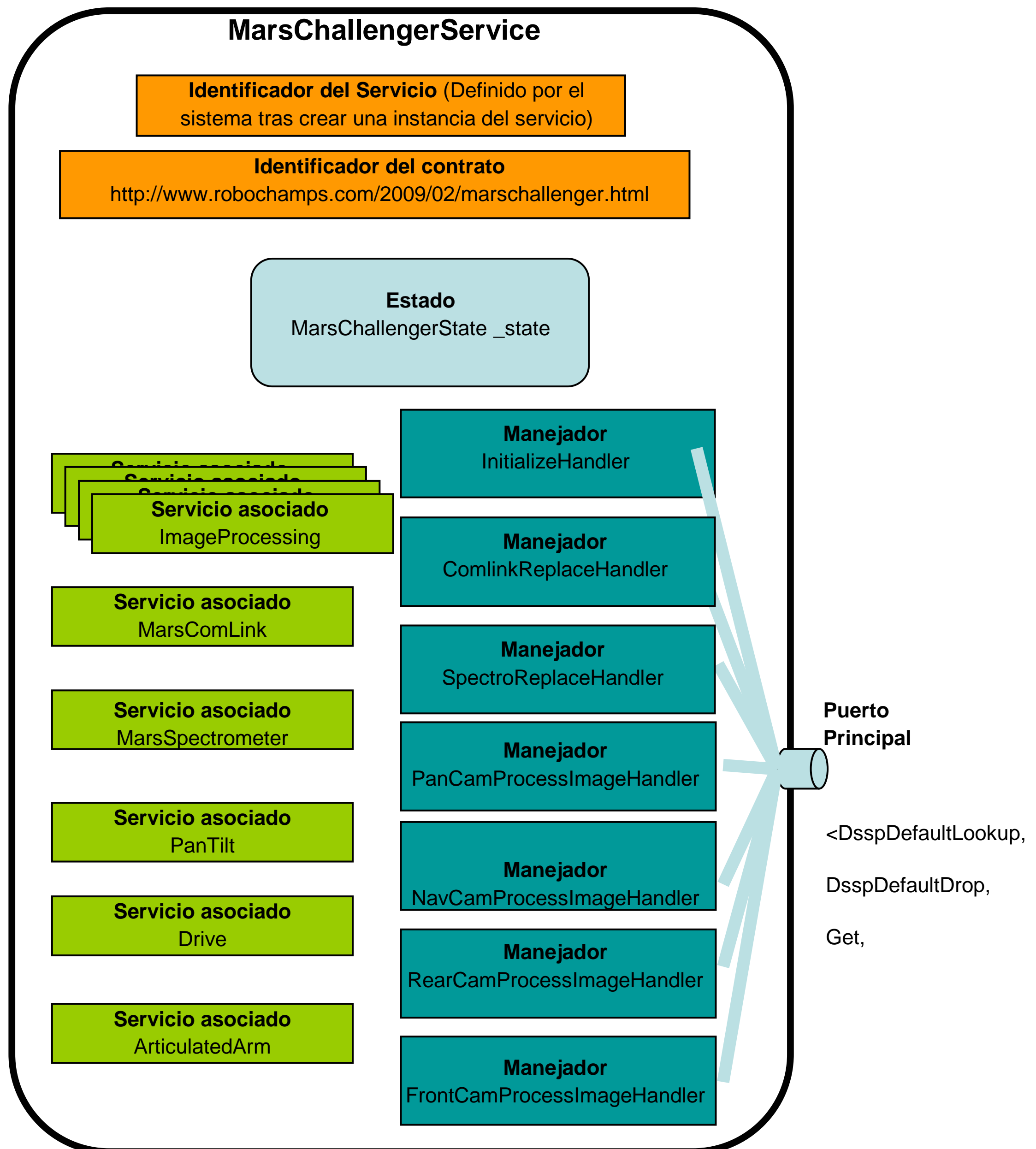

Figura 37.- Servicio MarsChallenger

A continuación, se comentan en mayor detalle los componentes de este servicio a través de la descripción de los ficheros que contienen el código que lo define.

5.2.1.- Fichero MarsChallengeTypes.cs

Nuevamente, las primeras líneas del código indican las librerías dinámicas que deben ser importadas, utilizando la directiva *using* y, a continuación, comienza la definición del ámbito del servicio. En este ámbito se define el identificador del contrato del servicio y dos elementos serializables:

- **MarsChallengerState**: clase que contiene la información relativa al estado del servicio. Contiene la siguiente información:
    - *RoverState State*: comportamiento actual del robot.
    - *MarsComLinkState ComlinkData*: información contenida en el puerto de comunicaciones del robot, que se corresponde con el estado del servicio que lo implementa.
    - *MarsSpectrometerState SpectrometerData*: estado del servicio que simula el comportamiento del espectrómetro del robot.
    - *ImageResult FrontCamData*: resultados obtenidos al realizar el último análisis de una imagen obtenida a través del servicio que simula la cámara frontal.
    - *ImageResult RearCamdata*: resultados del análisis del último fotograma obtenido a través del servicio de cámara trasera.
    - *ImageResult NavCamData*: instancia que encapsula la información extraída del análisis realizado sobre la última imagen capturada por el servicio de cámara de navegación.
    - *ImageResult PanCamData*: contiene los resultados del análisis realizado a la última imagen obtenida a través del servicio que simula el funcionamiento de la cámara panorámica del robot.
- **RoverState**: enumerado que contiene representa los distintos comportamientos del robot. Estos comportamientos son:
    - *Initialize*: se inicializan las distintas variables que maneja el robot. El robot no se desplaza.
    - *Navigate*: el robot navega por el entorno.
    - *EndMission*: el robot finaliza la misión.

Posteriormente, se declara el puerto principal del robot, definiendo las cinco operaciones que pueden ser solicitadas al servicio a través de él. Este puerto se encapsula en la clase denominada *MarsChallengerOperations* y permite mensajes del tipo *DsspDefaultLookup*, *DsspDefaultDrop*, *Get*, *Initialize*, *HttpGet*. Las dos primeras permiten solicitar, respectivamente, un listado de las operaciones permitidas por el servicio y la parada del mismo. La operación *Get* se utiliza para solicitar información sobre el estado del servicio, mientras que la operación *Initialize* realiza una petición de inicialización del estado del robot. Por último, mediante el envío de un mensaje del tipo *HttpGet* se realiza una solicitud del estado completo del robot con la finalidad de poder ser representado en un navegador Web.

Finalmente, se declaran las clases *Get* e *Initialize*, que encapsulan los mensajes del mismo nombre. La primera de ellas hereda el comportamiento de una clase predefinida del mismo nombre. La segunda, deriva de la clase *Update*, que, como se explica anteriormente, permite solicitar cambios en el estado del servicio. En este último caso, podemos comprobar que el cuerpo del mensaje no es ninguna clase predefinida, sino una clase que se declara en el mismo fichero, denominada *InitializeRequest*. Este hecho debe servir de ejemplo sobre cómo incluir nuevos tipos de datos en tipos de mensajes predefinidos.

5.2.2.- Fichero MarsChallenger.cs

En este fichero se encuentra definido el comportamiento del robot y conforma el bloque principal de código que será evaluado en la competición. Tras el enumerado de librerías dinámicas, se especifica, nuevamente, una breve descripción del servicio, así como el nombre que será mostrado del mismo y el identificador del contrato que le corresponde. A continuación, comienza la definición del servicio en su totalidad. En las primeras líneas, se incluyen las distintas variables globales de la clase. Estas variables serán utilizadas por los distintos algoritmos. No se incluye una descripción detallada de las mismas en este apartado, pues serán explicadas posteriormente en los diferentes algoritmos.

Este fichero puede dividirse en distintos apartados que corresponden con la definición de los componentes del servicio, así como las distintas operaciones que son llevadas a cabo por él.

5.2.2.1.- Estado del servicio

El estado del servicio se encuentra definido por una instancia de la clase *MarsChallengerState*. Por tanto, toda la información referente al estado del servicio debe encontrarse reflejada en dicha instancia. De manera similar a lo expuesto en el apartado 5.2.1, las líneas de código que definen el estado del servicio son las siguientes:

```
[ServiceState]
private MarsChallengerState _state = new MarsChallengerState();
```

5.2.2.2 Puerto Principal

En este caso, el puerto principal se define de la siguiente manera:

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

Como podemos observar, el elemento *_mainPort* representará el puerto principal del servicio, a través del cual los servicios asociados podrán realizar solicitudes a éste. Podemos observar que, en este caso, el atributo AllowMultipleInstances tiene asignado valor false. Esto implica que, aunque distintos servicios se asocien al nuestro, sólo existirá una instancia del mismo y, por tanto, el identificador del puerto principal que éstos utilicen para comunicarse con el servicio MarsChallengerService será el mismo en todos los casos.

5.2.2.3.- Servicios asociados

Los servicios asociados a nuestro servicio son los siguientes:

- **MarsComLink**: proporciona la funcionalidad atribuida al puerto de comunicaciones del robot. Su puerto principal lo representa la variable *_comlinkPort* y su puerto de suscripción la variable *_comlinkNotify*.

- **MarsSpectrometer**: simula el comportamiento del espectrómetro del robot. Su puerto principal se accede a través de la variable *_spectroPort* y su puerto de suscripción a través de la variable *_spectroNotify*.

- **ImageProcessing**: lleva a cabo las tareas de procesado de imagen. El servicio *MarsChallenger* se encuentra asociado a cuatro instancias distintas del servicio *ImageProcessing*, una por cada una de las distintas cámaras del robot. Por tanto, los puertos principales y de suscripción para las servicios de procesado de imágenes obtenidas a través de las cámaras panorámica, de navegación, frontal y trasera son, respectivamente, *_pancamPort*, *_pancamNotify*, *_navcamPort*, *_navcamNotify*, *_frontcamPort*, *_frontcamNotify*, *_rearcamPort* y *_rearcamNotify*.

- **Drive**: gestiona el movimiento de las ruedas del robot. Sólo posee implementada la solicitud *SetDrivePower*, mediante la cual, es posible indicar la potencia que debe asignársele a las ruedas de cada uno de los lados del robot. Su puerto principal es *_drivePort*, pero no se especifica puerto de suscripción, pues no se realiza suscripción a este servicio.

- **PanTilt**: representa la cabeza del robot y permite efectuar rotaciones horizontales y verticales en la misma. Su puerto principal se accede a través de la variable *_headPort*, pero no se precisa puerto de suscripción.

- **ArticulatedArm**: proporciona la funcionalidad correspondiente al brazo mecánico del robot. El puerto principal está representado por la variable *_armPort*.

5.2.2.4.- Manejadores

Los manejadores para los distintos tipos de mensajes y notificaciones recibidos son los siguientes:

- **InitializeHandler**: está encargado de procesar las solicitudes de inicialización del servicio, por lo que en su ejecución se inicializan las distintas variables del servicio que controla al robot, así como habilitar los distintos receptores de la siguiente manera:

```
MainPortInterleave.CombineWith(
  Arbiter.Interleave(
    new TeardownReceiverGroup(),
    new ExclusiveReceiverGroup
                (
Arbiter.Receive<comlink.Replace>(true, _comlinkNotify, ComlinkReplaceHandler),

Arbiter.Receive<spectro.Replace>(true, _spectroNotify, SpectroReplaceHandler),

Arbiter.Receive<processor.ProcessImage>(true, _pancamNotify,
PanCamProcessImageHandler),

Arbiter.Receive<processor.ProcessImage>(true, _navcamNotify,
NavCamProcessImageHandler),

Arbiter.Receive<processor.ProcessImage>(true, _frontcamNotify,
FrontCamProcessImageHandler),

Arbiter.Receive<processor.ProcessImage>(true, _rearcamNotify,
RearCamProcessImageHandler)
                    ),
                    new ConcurrentReceiverGroup()));
```

Como podemos observar, los receptores habilitados son del tipo exclusivo, es decir, mientras uno de ellos esté ejecutándose, ningún otro puede ejecutarse.

- **ComlinkReplaceHandler**: recibe notificaciones sobre el nuevo estado del servicio que simula el puerto de comunicaciones, con lo que reemplaza la información de la variable *ComlinkData*, almacenada en el estado del servicio.

- **SpectroReplaceHandler**: maneja las notificaciones de cambio de estado del servicio que controla el espectrómetro. Al recibirlas, actualiza el estado del servicio modificando la variable *SpectrometerData*

- **PanCamProcessImageHandler**: se ejecuta cada vez que se obtienen nuevos resultados derivados de analizar una imagen proveniente del servicio de cámara panorámica. Por tanto, actualiza el estado del servicio incorporando estos resultados a la variable *PanCamData*.

- **NavCamProcessImageHandler**: recibe los resultados del último análisis realizado a una imagen de la cámara de navegación y los incorpora al estado del servicio actualizando la variable *NavCamData*.

- **FrontCamProcessImageHandler**: se ejecuta cuando se obtiene una notificación por parte del servicio de cámara frontal informando acerca de los resultados del análisis a una nueva imagen captada por dicha cámara. Esta información se incorpora al estado del servicio a través de la variable *FrontCamData*.

- **RearCamProcessImageHandler**: recibe los resultados del último análisis realizado a una imagen captada por el servicio de cámara trasera. Por tanto, actualiza el estado actual del servicio modificando la variable *RearCamData*.

5.2.2.5.- Lógica de ejecución

Nuevamente, para comenzar la ejecución del servicio, se invoca al método Start. Este método envía un mensaje del tipo *Initialize* a través del puerto principal, provocando, por tanto, que se ejecute el manejador *InitializeHandler*. Mediante este manejador, se inicializan las distintas variables y receptores del servicio. Además, se invoca al método *RearmMainLoop*. Este método habilita un receptor no persistente, es decir, que se ejecuta una única vez, asociado a un puerto del tipo *TimeoutPort*. Este puerto implementa un contador de cuenta atrás, lo que provoca que se reciba un mensaje, con el tiempo del sistema, transcurrido un tiempo especificado en su creación. Así pues, se consigue que comience a ejecutarse el método *MainLoopHandler*. Este método se ejecuta iterativamente a través de llamadas a *RearmMainLoop* y su cometido es indicar a los servicios que componen el robot que realicen una serie de acciones en base al estado actual del servicio de control. En concreto, en base al valor de la variable *_state.State*.

En un principio, el comportamiento del robot consiste en avanzar diez metros en línea recta e, inmediatamente después, transitar el comportamiento de fin de misión, deteniéndose y dando ésta por terminada.

5.2.2.6.- Funciones Auxiliares

Finalmente, se incluyen dos funciones cuya finalidad es llevar a cabo tareas de soporte a otros algoritmos. Estas funciones son:

- **ToRadians**: que convierte una cantidad especificada en grados a su correspondiente representación en radianes.

- **Distance**: permite obtener la distancia geométrica entre dos puntos, especificadas las tres coordenadas de ambos.

Capítulo 6:
Solución Implementada

Para poder elaborar una solución al problema expuesto en el apartado 4, resulta imprescindible obtener, por un lado, los conocimientos relativos a la programación MRDS que se describen en el capítulo 3 y, por otro, el trasfondo relacionado con el tema que se introdujo en el apartado 2 con el estado del arte de la navegación robótica.

En este capítulo se describe la solución que hemos implementado y presentado a la competición. Para ello, se relatan los distintos pasos que hemos realizado para implementar esta solución y las razones por las que tomamos todas y cada una de las decisiones de implementación.

6.1.- Navegación Global

El problema inicial que hemos de resolver implica conseguir que el robot se desplace desde la posición en que se encuentra, sea esta cual sea, hasta la siguiente área de interés, siendo éste un problema de navegación global. Como sistema de soporte a este proceso, es posible obtener, en todo momento, las coordenadas exactas que definen la posición del robot. Para ello, se accede a la variable *_state.ComLinkData.Position*, que forma parte del estado del servicio, y que se actualiza con la información obtenida a través del puerto de comunicaciones. Este dispositivo, como se indica anteriormente, posee un sistema de localización global GPS.

Por otro lado, a través del puerto de comunicaciones también es posible conocer la orientación del robot mediante sus coordenadas eulerianas o ángulos de navegación *Yaw, Pitch* y *Roll*:

- **Yaw o guiñada**: rotación con respecto al eje vertical o Y.
- **Pitch o cabeceo**: rotación con respecto al eje X.
- **Roll o alabeo**: rotación con respecto al eje Z.

Estas rotaciones del robot se muestran en la siguiente imagen:

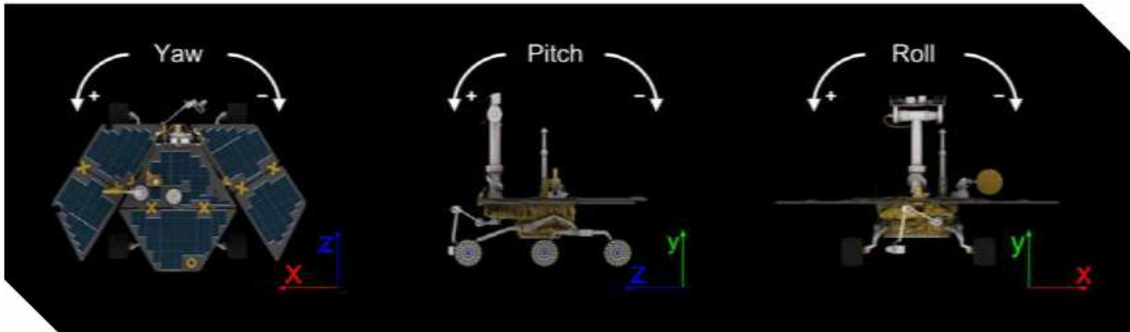

Figura 38.- Coordenadas Yaw, Pitch y Roll

Estas coordenadas se definen por el siguiente sistema de valores:

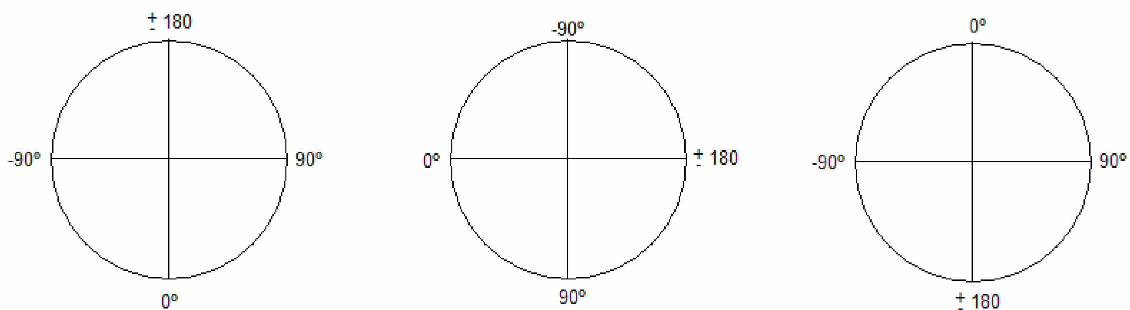

Figura 39.- Sistema angular de valores de las coordenadas Yaw, Pitch y Roll, respectivamente

Por tanto, si se diese la situación en que el valor de *Yaw* es 0º, el de *Pitch* 20º y el de *Roll* 70º, esto implicaría que el robot se encuentra orientado en la dirección negativa del eje Z, que su cabeceo está inclinado hacia abajo y que se encuentra peligrosamente ladeado hacia la derecha.

Finalmente, a través del puerto de comunicaciones también se puede obtener la lista de localizaciones de interés en las que se debe realizar el análisis de muestras de roca. Estas localizaciones se encuentra ordenada, es decir, las seis posiciones que se incluyen corresponden, por este orden, a las áreas de interés de la uno a la seis. Así pues, en primera instancia se posee la información sobre la localización exacta del robot mediante un sistema de seis coordenadas, *(X,Y,Z, Yaw, Pitch, Roll)*, y una lista de localizaciones a las que el robot debe desplazarse.

Por tanto, la primera acción que el robot debe realizar es averiguar cuál es su siguiente destino. Para ello, se declara la variable global *targetNum*:

```
private int targetNum = 0;
```

Esta variable indica cuál es la localización de interés hacia la que el robot se dirige. Se inicializa para que apunte a la primera localización de la lista y su valor se incrementa a medida que se visitan las diferentes zonas de interés. A través de la variable *targetNum*, es posible obtener la siguiente localización a la que el robot debe desplazarse, por lo que, una vez obtenida la información respectiva a la localización del robot, puede comenzar la navegación.

Como se comentó anteriormente, el comportamiento del robot está definido el estado del mismo mediante la variable *_state.State*. Así pues, para que el robot se desplace de un punto a otro se debe poseer el comportamiento *Navigate*. Una vez que el robot se ha desplazado hasta el área de interés, se precisa que localice una muestra de roca, por lo que se debe transitar al comportamiento *RockFinder*. Por último, durante el desplazamiento del robot, es posible que éste deba evitar algún obstáculo en su camino variando su ruta, por lo que, en ese momento, transitará al comportamiento *Stuck*.

6.1.1.- Comportamiento Navigate

Como se muestra en el apartado 2.1, para que el robot se desplace desde una localización a otra puede resultar útil poseer un mapa del entorno. Sin embargo, esta técnica resulta más apropiada para casos en los que el robot se encuentra en un escenario cerrado. Por tanto, hemos decidido implementar la navegación sin utilizar un mapa del entorno. Para ello, y basándonos en las distintas teorías de elaboración de caminos, así como el algoritmo *WedgeBug* expuesto en el capítulo 2, hemos implementado un procedimiento mediante el cual el robot generará un camino desde su posición hasta el área de interés. Para elaborar ese camino, se poseen las coordenadas de la posición actual del robot y un punto que se encuentra dentro del área de interés al que se desea llegar.

En lo que respecta a la representación del camino a seguir, éste constará de una serie de puntos por los que el robot deberá transitar. Para ello, debe implementarse el procedimiento *getPath(origen, destino)*. Mediante este procedimiento, se generará un camino en el que los distintos puntos que lo conforman distan entre sí distancias similares. Estos puntos se almacenan en un array cuya última posición contiene la posición destino. En el código, el camino a seguir se encuentra almacenado en la variable *Path*. A la hora de determinar las distintas posiciones, sólo se tendrá en cuenta las coordenadas X y Z, pues el robot sólo se desplaza sobre el terreno y la verticalidad se obvia.

El código de este algoritmo se muestra a continuación:

```
private ArrayList getPath(comlink.Position from, comlink.Position to)
        {
            ArrayList result = new ArrayList();

            //Every step will move (at most) ten meters on X and Z axis
```

```
            int steps = (int)(Distance(from, to) / 10); //Number of
intermediate steps
            int xlength = (int) (Math.Abs(from.X - to.X));//Distance between
every step in X axis
            int zlength = (int)(Math.Abs(from.Z - to.Z));//Distance between
every step in Z axis
            bool fixedX = false;
            bool fixedZ = false;

            if (steps == 0)
                result.Add(to);
                return result;
            }

            if (xlength < 5) {
//If X distance is little, move directly to that coordinate value
                fixedX = true;
            }

            if (zlength < 5) {
//If Z distance is little, move directly to that coordinate value
                fixedZ = true;
            }


            if (!fixedX) { xlength /= steps; }

            if (!fixedZ) { zlength/= steps; }


            //Calculate the position of every step and add it to the path
            for (int i = 0; i < (steps-1); i++){
                comlink.Position point = new comlink.Position();
                if (!fixedX) {
                    if (to.X > from.X) {
                        point.X = from.X + (i + 1) * xlength;
                    }else{
                        point.X = from.X - (i + 1) * xlength;
                    }
                }else{
                    point.X = to.X;
                }

                if (!fixedZ) {
                    if (to.Z > from.Z){
                        point.Z = from.Z + (i + 1) * zlength;
                    }
                    else{
                        point.Z = from.Z - (i + 1) * zlength;
                    }
                } else  { point.Z = to.Z;
                }
             result.Add(point);
            }
            comlink.Position finish = new comlink.Position();
//Final step: Destination
            finish.X = to.X;
            finish.Z = to.Z;
            result.Add(finish);
            return result;
        }
```

Supongamos que el robot debe desplazarse desde la posición *(0,0,0)* hasta la posición *(-33, 0, -30)*. La distancia entre ambos puntos es $\sqrt{((-33-0)^2+(20-0)^2)}$, que equivale aproximadamente a 39 metros. Así pues, se generarán *(39/10) = 3* pasos entre el origen y el destino. En cada paso, se procura mantener la distancia recorrida en la dirección de los ejes X y Z, por lo que en este caso los distintos puntos distarán *(33/3)* metros en el eje X y *(20/3)* metros en el eje Z.

Finalmente, al camino elaborado se añade la posición destino como un paso más, con lo que el robot terminará su camino en dicha posición. Por tanto, el camino generado, en coordenadas *(X, Z)* será el siguiente:

(-11,-6.66)	(-22,-13.33)	(-33,-20)

Figura 40.- Camino generado

Teniendo en cuenta que el robot no es un punto en el espacio, sino que posee unas determinadas dimensiones, no es necesario que el robot se localice exactamente en una posición del camino para considerar que ésta se ha alcanzado. En cambio, hemos considerado que el robot ha alcanzado una determinada posición cuando se encuentra en un área próxima. El tamaño de esta área dependerá de si se trata de una posición intermedia o la posición destino. En el caso de que se trate de una posición intermedia, se considera que el robot la ha atravesado siempre que se encuentre a menos de tres metros de la misma. Por otro lado, en el caso de que se trate de la posición destino, el robot deberá encontrarse a menos de un metro de dicha posición para que el camino se considere recorrido. Este sistema de determinación de distancias se encuentra codificado en el código de la siguiente manera:

```
if (
(pathNum<(path.Count-1)&&(Math.Abs(horizontalD)>3||Math.Abs(verticalD) > 3))
||
(pathNum>=(path.Count-1)&&(Math.Abs(horizontalD)>1||Math.Abs(verticalD) > 1))
){//El robot no ha alcanzado su destino  }
```

Siendo *horizontalD* y *verticalD* las distancias actuales, con respecto a la siguiente posición destino, en los ejes X y Z, respectivamente.

En la siguiente imagen se muestra un ejemplo del camino diseñado por el robot entre dos posiciones, así como las áreas en las que se considera una posición como visitada:

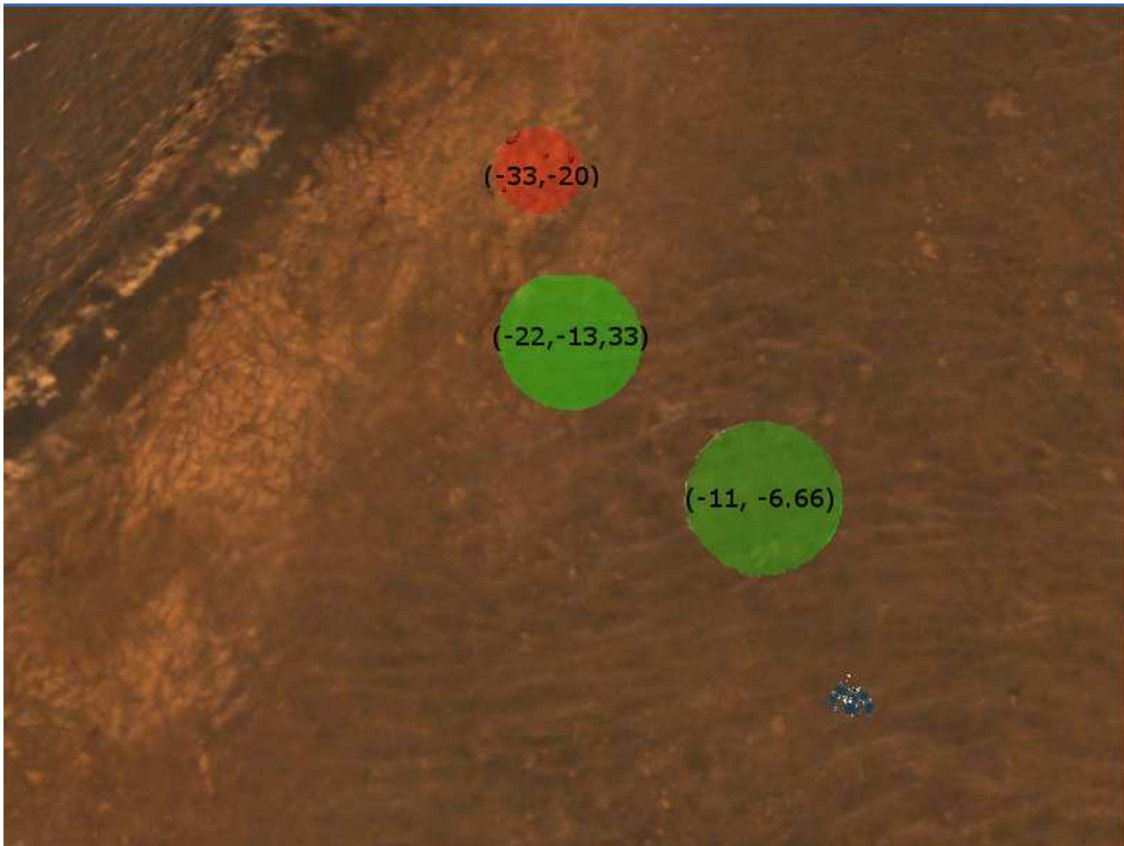

Figura 41.- Representación sobre el escenario del camino elaborado por el robot

Como podemos comprobar, en esta situación, el robot diseñaría un camino en tres pasos hacia la zona de interés contenida en el interior del área roja. Además, como se comenta anteriormente, cada vez que el robot ingrese en una de las áreas coloreadas, se considera un paso como tomado, con lo que su siguiente localización destino pasa a ser el siguiente paso del camino diseñado.

Con el objetivo de simplificar los cálculos, hemos incluido un mecanismo para evitar realizar operaciones innecesarias. Dichas operaciones, por ejemplo, pueden aparecer en un caso en el que el robot deba desplazarse desde la posición (-100, 20) a la posición (-200, 19). Dada esta situación, la distancia aproximada es de 100 metros, requiriendo tomar 10 pasos. Si no se aplicase el mecanismo de simplificación, las distintas posiciones por las que el robot transita incluirían las siguientes coordenadas en Z: (19.9, 19.8, 19.7, … 19.1, 19). Para obtener cada una de estas coordenadas deberían realizarse operaciones decimales, lo que precisa una cantidad relativamente elevada de recursos computacionales. Sin embargo, el mecanismo implementado implica que, si la siguiente distancia, con respecto a un eje, que se avanza es inferior a cinco metros, no se generen pasos intermedios entre la posición actual y la de destino.

Por tanto, en este caso el camino sería:

(-110,19)	(-120,19)	(-130,19)	(-140,19)	(-150,19)	(-160,19)	(-170,19)	(-180,19)	(-190,19)	(-200,19)

Figura 42.- Camino elaborado

Una vez que se ha elaborado el camino a transitar, es necesario que el robot realice las acciones necesarias para desplazarse a cada uno de los puntos de dicho camino. Para ello, el primer paso consiste en determinar la posición relativa del siguiente punto con respecto a sí mismo. Esta posición relativa se corresponde con las combinaciones de los cuatro puntos cardinales: Norte, Noroeste, Oeste, Suroeste, Sur, Sudeste, Este y Noreste. Esta posición relativa se consigue a través de la obtención de la distancia entre el robot y el punto destino con respecto a los ejes *X* y *Z*.

Por tanto, se pueden dar dos situaciones:

- Si el robot no está orientado hacia el punto cardinal que corresponde a la posición relativa en que se encuentra el punto al que se dirige, éste realiza un movimiento de rotación que lo sitúe en la orientación adecuada.

- Si se encuentra orientado en la misma dirección, se realiza un proceso de orientación más preciso, con el fin de que el robot se desplace en la dirección adecuada.

En el primer caso, la rotación se realiza de distinta manera dependiendo de la orientación que se desee conseguir. Si se desea que el robot se oriente hacia el Norte, Sur, Este u Oeste, se realiza una llamada al método *lookUp*, *lookDown*, *lookRight* o *lookleft*, respectivamente. De esta manera, el robot rotará hasta orientarse en la dirección deseada, con una precisión de 5 grados. Por ejemplo, si la localización destino se encuentra inmediatamente al sur del robot, la posición relativa será Sur. Si el robot no está orientado en esta dirección, correspondiente al valor 0 de *Yaw*, se invoca el método *lookDown*(). Este método hará que el robot rote sobre sí mismo en el sentido que más rápidamente permita obtener un valor de *Yaw* comprendido en el intervalo [-5, 5].

Por otro lado, si la localización destino está en dirección Noroeste, Noreste, Sudeste o Suroeste, el proceso de orientación se realiza en dos pasos. Primero, se hace rotar al robot hasta que se encuentre orientado en la dirección deseada y, a continuación, se rota al robot más despacio hasta conseguir una orientación próxima a la de la recta que une su posición y el destino.

Supongamos que el robot se encuentra en la posición (0, 0, 0) y que el próximo paso a realizar debe situarlo en la posición (10, 0, 10). Por tanto, la orientación relativa del próximo destino es Noreste.

Si el valor de la coordenada Yaw del robot es 0º, la situación es la siguiente:

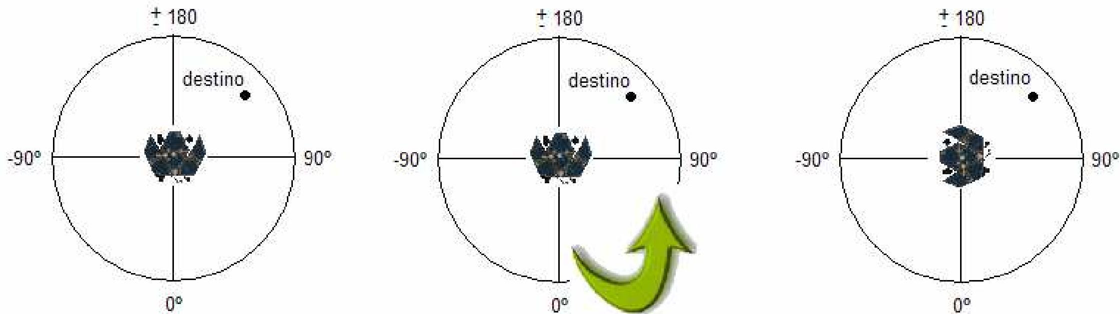

Figura 43.- Sentido de giro del robot

El robot gira en sentido antihorario hasta que su orientación es Noreste, es decir, el valor de Yaw pertenece al intervalo (90, 180). Sin embargo, esta rotación no es suficiente, pues se precisa una orientación cuyo Yaw sea 135º, lo que dista bastante de valores como 91º o 179º que podrían ser tomados por aceptables si sólo se tiene en cuenta el punto cardinal hacia el que el robot está orientado.

Para realizar un proceso de orientación más preciso, una vez que el robot esté orientado hacia el cuadrante deseado, dicho cuadrante se divide en tres áreas de la siguiente manera:

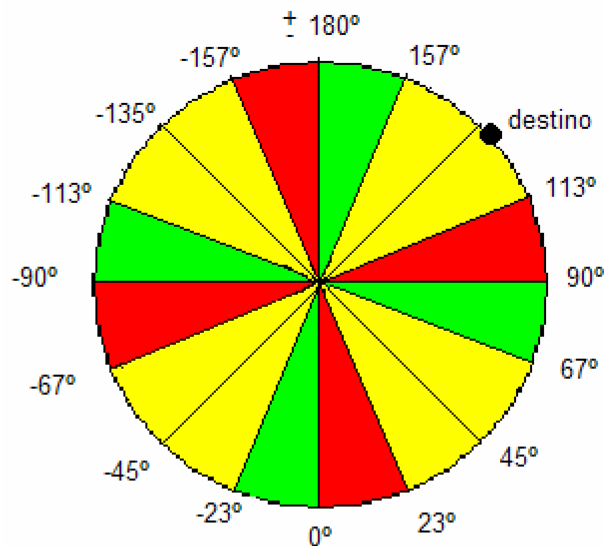

Figura 44.- División en áreas de los cuadrantes

Así pues, se definen tres nuevos tipos de orientación en cada cuadrante. Para el cuadrante superior derecho, por ejemplo, existe la orientación situada en (90º,113º), la orientación para destinos situados en [113º, 157º] y la última orientación en (157º, 180º). De esta manera, en el caso anterior, no bastaría con que el robot esté orientado hacia el Noreste, sino que su coordenada Yaw debe encontrarse en el intervalo [113º, 157º].

Para conocer cuál de las tres orientaciones se debe proporcionar al robot, se utiliza la pendiente de la recta formada entre el punto actual del robot y el punto destino. De esta manera, los intervalos se corresponden con las siguientes pendientes:

- Pendiente en (0, 0.5] ➔ orientación verde.
- Pendiente en (0.5, 2) ➔ orientación amarilla.
- Pendiente en [2, +∞)➔ orientación roja.

Podemos observar, que las porciones en las que se dividen los cuadrantes no son del mismo tamaño. El objetivo de esta división irregular es simplificar los cálculos, pues si la división fuese regular, no podrían utilizarse los valores de la pendiente que se indican un poco más arriba, sino otros como π/2, que implicarían realizar operaciones de coma flotante. A continuación, se incluye el código que implementa todo este proceso, en el caso de que la posición destino se encuentra al Sudeste de la posición actual. Para el resto de orientaciones el código es similar:

```
if (horizontalD > 0 && verticalD < 0){
    //Destino en una posición al sudeste
    spinRight = true;

    if (Math.Abs(_state.ComLinkData.Yaw) > 135 || _state.ComLinkData.Yaw > 90)
    {
           rightWheel = -0.25;
           leftWheel = 0.25;
    }
    else if (_state.ComLinkData.Yaw < 0)
         {
              rightWheel = 0.25;
              leftWheel = -0.25;
         }
         else
         {
              double slope = Math.Abs(verticalD / horizontalD);
              if (slope >= 0.5 && slope <= 2)
              {
               if (_state.ComLinkData.Yaw >= 0 && _state.ComLinkData.Yaw < 23)
               {
                     rightWheel = 1;
                     leftWheel = 0.8;
               }
        else if (_state.ComLinkData.Yaw >= 67 && _state.ComLinkData.Yaw <= 90)
         {
                rightWheel = 0.8;
                leftWheel = 1;
         }
```

```
      }
    else if (slope < 0.5)
         {
          if (_state.ComLinkData.Yaw >= 67 && _state.ComLinkData.Yaw <= 90)
           {
             rightWheel = -0.25;
             leftWheel = 0.25;
           }
        else if (_state.ComLinkData.Yaw >= 45 && _state.ComLinkData.Yaw <= 67)
          {
              rightWheel = 0.8;
              leftWheel = 1;
          }
     }
     else if (slope > 2)
       {
         if (_state.ComLinkData.Yaw >= 0 && _state.ComLinkData.Yaw <= 45)
          {
            rightWheel = 0.25;
            leftWheel = -0.25;
          }
       else if (_state.ComLinkData.Yaw >= 45 && _state.ComLinkData.Yaw <= 67)
         {
           rightWheel = 1;
           leftWheel = 0.8;
         }
       }
      }
     }
```

6.1.2.- Comportamiento RockFinder

El robot, tras desplazarse hasta el área de interés indicado, debe localizar las muestras de roca que debe analizar. Estas muestras se encuentran en las proximidades del robot, por lo que los sistemas de navegación global, como GPS, carecen de utilidad. Sin embargo, el robot cuenta con otro tipo de sensores mucho más útiles en lo que a navegación local se refiere. Estos sensores son las distintas cámaras que el robot posee y a través de ellas, como veremos más adelante, es posible detectar las muestras de roca analizar.

La primera acción que el robot realiza el transitar a este comportamiento consiste en colocar el brazo mecánico en posición de análisis. Esta posición facilita que el espectrómetro instalado en el robot entre en contacto con una muestra de roca a medida que el robot se aproxime a ésta. Para colocar el brazo mecánico en posición de análisis, se invoca el método *setFindingPosition()*, que provocará el envío de varios al servicio que controla el brazo articulado. Para realizar la acción inversa, es decir, para que el brazo abandone la posición de análisis y se coloque de manera que no entorpezca el desplazamiento del robot, se invoca al método *setNormalPosition()*. A continuación, se comienzan a tener en cuenta los análisis realizados a las imágenes obtenidas a través de la cámara de navegación. Estos análisis son realizados por el servicio *ImageProcessing* y consisten en analizar todos los píxeles de la imagen obtenida a través de esta cámara para localizar aquellos que representan colores que sólo aparecen en las rocas que se desea analizar.

Además de comprobar que hay muestras de roca a la vista, se trata de detectar la posición relativa de dichas muestras con respecto al robot. Para ello, se realizan diez divisiones en la imagen, cinco verticales y cinco horizontales:

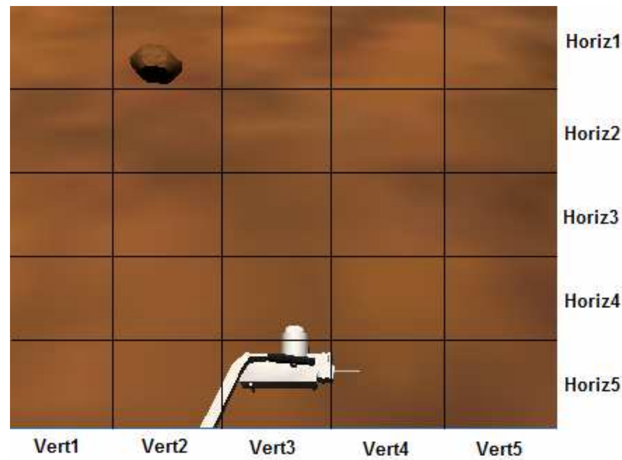

Figura 45.- División cuadricular de una imagen

Así pues, cada vez que se detecta un píxel correspondiente a una roca, hecho que se cumple si el valor RGB de dicho píxel se encuentra entre los valores RGB (21,11,4) y (32, 19, 10), además de aumentarse un contador que indica la cantidad de píxeles de roca identificados, se localiza ese píxel en dos de las particiones realizadas en la imagen: en una de las verticales y en una de las horizontales.

Por último, se declaran cinco zonas en la imagen. Estas cinco zonas se corresponden con las posiciones relativas de las rocas con respecto al robot y es la información que le llegará al servicio de control *MarsChallenger* a través del parámetro *RockPosition* de la clase *ImageResult* que éste almacena en su estado como parámetro *_state.NavCamData*. Las cinco zonas en las que se divide la imagen y los movimientos que realiza el robot al detectar rocas en ellas son los siguientes:

- **Zona 1**: la roca se localiza en la parte inferior izquierda de la imagen. El robot retrocede.

- **Zona 2**: la roca se localiza en la parte superior izquierda. El robot rota hacia la izquierda.

- **Zona 3**: hay una roca justo delante del robot. El robot avanza en línea recta.

- **Zona 4**: la roca se localiza en la parte superior derecha de la imagen. El robot gira hacia la derecha.

- **Zona 5**: se detecta una roca en la parte inferior derecha de la imagen. El robot retrocede.

En la siguiente imagen se muestra esta división en zonas:

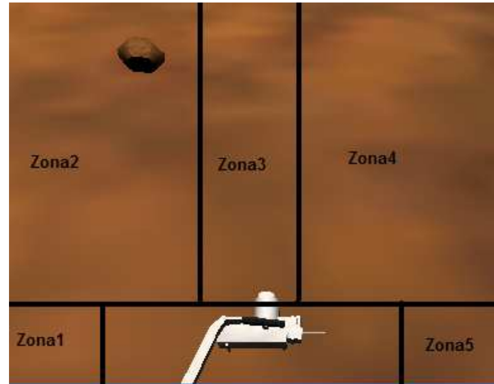

Figura 46.- División en zonas de un fotograma

Como se comentó anteriormente, el objetivo de este comportamiento es localizar y analizar las muestras de roca, por lo que, tras colocar el brazo articulado en posición de análisis, el robot realiza un rastreo de las rocas en sus alrededores. Para ello, el robot gira su cabeza hacia debajo de tal manera que enfoca directamente el terreno que el robot tiene delante de él. Si no se encuentra ninguna roca, el robot vuelvo a levantar la cabeza para buscar las rocas a más distancia. Si continúa sin ver ninguna roca, realiza un giro en torno a sí mismo hasta que avista alguna roca, momento en el que avanza y vuelve a bajar la cabeza.

Este procedimiento de rastreo y análisis de rocas se muestra en la siguiente imagen de izquierda a derecha y de arriba a abajo:

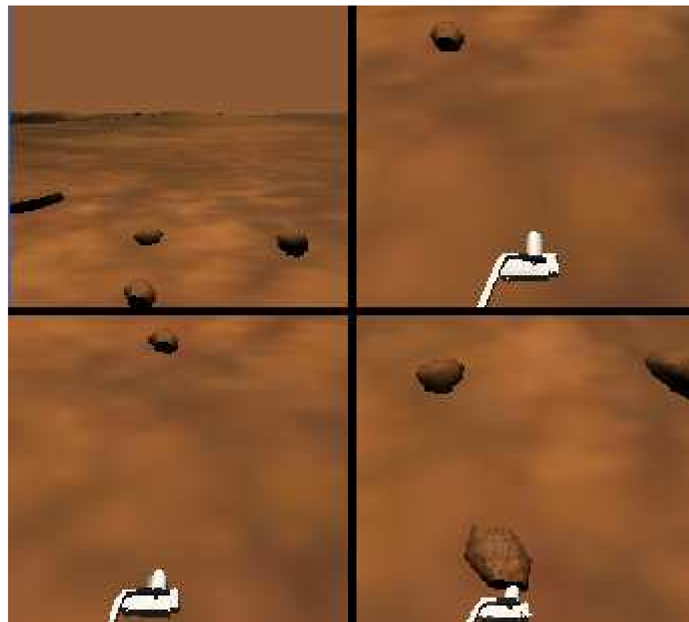

Figura 47.- Proceso de análisis de una muestra

Si tras cierto tiempo rotando el robot no encuentra ninguna o roca, o si el robot abandona el área de interés en el que debe localizar las rocas, se transita al estado *Navigate* con el objetivo de volver a aproximarse de nuevo a las rocas. Cuando una nueva roca es analizada, el servicio que controla el espectrómetro envía un mensaje de notificación al servicio MarsChallenge, por lo que el robot toma conocimiento de este hecho y elabora un nuevo camino hacia la siguiente área de interés.

6.2.- Navegación Local: Mecanismos de soporte a la navegación

En los anteriores apartados se han descrito los procedimientos mediante los cuales el robot se desplaza entre las distintas localizaciones de interés y analiza las rocas situadas en éstas. Sin embargo, todo ese proceso tendría grandes probabilidades de fracaso a causa de acontecimientos desfavorables no previstos, como que las ruedas del robot se atasquen en un obstáculo o que el robot vuelque debido a su tránsito por una pendiente muy pronunciada. Con el fin de evitar las consecuencias desfavorables de estas situaciones, se ha diseñado una serie de mecanismos de prevención de atasco y vuelco. Por otro lado, debido a que el terreno sobre el que se encuentra el robot es irregular, es posible que se produzcan deslizamientos que desvíen al robot del camino planificado, por lo que también se incluyen mecanismos de corrección de ruta.

6.2.1.- Mecanismos de prevención de atasco

En el terreno existen varios obstáculos con los que pueden provocar que el robot quede atascado, como las rocas y la arena del fondo del cráter. Sin embargo, en los algoritmos de navegación anteriormente comentados, este hecho no se tiene en cuenta, pues estos obstáculos no son tan visibles como lo sería una pared. En el caso de la arena, es posible viajar a través de ella, pero no permite rotar, pues las ruedas resbalan en ella. En el caso de las rocas, no son un obstáculo que haya que evitar, sino más bien un elemento al que hay que aproximarse, pero existen casos en los que el robot atasca sus ruedas en ellas al intentar girar. Además, es posible que el robot circule por encima de una roca sin problemas. Por tanto, el comportamiento a implementar no consistirá en detectar y evitar los obstáculos, pues estos no impiden el paso del robot, sino un mecanismo mediante el cual el robot sea capaz de detectar que está atascado para, antes de continuar su camino, realizar ciertos movimientos que lo desatasquen.

Para realizar esta detección, se ha tenido en cuenta el hecho más destacable de una situación de atasco: la ausencia de desplazamiento a pesar del movimiento de las ruedas. Así pues, en cada iteración, se comprueba si el robot se ha desplazado o rotado sobre sí mismo. Si el robot ni se ha desplazado ni ha rotado y sus ruedas están en movimiento, se aumenta un contador denominado *StopCounter*. Si este contador alcanza un valor igual a cincuenta, se considera que el robot está atascado, se almacena la posición en la que este atasco se produjo en la variable *LastStuck* y se transita al comportamiento *Stuck*.

En este comportamiento, se comprueba si es la primera vez que el robot atasca en una posición muy próxima a la que se encuentra en ese momento. Si es la primera o segunda vez consecutiva que el robot se atasca en ese lugar, el robot realiza un movimiento de retroceso y giro que lo libere de su atasco. Si, por el contrario, el robot se ha atascado en ese lugar tres o más veces, se invoca el método *getPathAvoidStuck*, que recalcula una nueva ruta para el robot en la que se evita pasar por las proximidades de esa localización.

6.2.2.- Mecanismos de prevención de vuelco

En el centro del escenario, se sitúa el cráter Endurance. Para acceder y salir de este cráter, es necesario que el robot se desplace por terrenos con pendientes muy pronunciadas que pueden ocasionar que éste vuelque. Si esta situación de vuelco se produjese, la misión se abortaría, pues el robot no posee mecanismos para recobrar su posición normal. Por tanto, se han implementado una serie de medidas con el objetivo de evitar que este vuelco se produzca. La primera de ellas se fundamenta en el hecho de que, si el robot, mientras desciende por una pendiente, realiza una disminución brusca de su velocidad o, incluso, retrocede, sus ruedas traseras pierden la sujeción al terreno, por lo el robot vuelca adelante. Así pues, se incluye un mecanismo que de manera proactiva evita una disminución brusca de velocidad cuando el valor de la coordenada *Pitch* sea superior a 15º, realizando, en su lugar, esta disminución escalonadamente. Por otro lado, también es posible que el robot vuelque si acelera bruscamente al subir una pendiente. En este caso, existe un mecanismo, similar al anterior, que disminuye la aceleración del robot, o incluso lo hace retroceder, cuando el valor de la coordenada *Pitch* es inferior a -35º.

Finalmente, se incluye una pequeña excepción en el algoritmo de elaboración de rutas, pues resulta muy complicado que el robot sea capaz de abandonar el cráter por su cuenta una vez que ha entrado en él. En la vida real, esta tarea requirió más de dos semanas para ser realizada y la solución adoptada fue indicar al robot en qué punto del cráter la pendiente era menos pronunciada[4]. Por tanto, se incluye un mecanismo para detectar que el robot se encuentra dentro del cráter y desea salir para, ante esta situación, elaborar una ruta que permita al robot abandonar el cráter por la zona indicada.

---

[4] Ver apartado "*sol 292-298, December 06, 2004: Edging Out of 'Endurance'*" en http://marsrovers.jpl.nasa.gov/mission/status_opportunityAll_2004.html#sol292

6.2.3- Sistemas de corrección de ruta

Como se comenta anteriormente, es posible que el robot abandone la ruta planificada debido a causas externas, como que derrape por una pendiente. Al desviarse de la ruta es posible que se den dos situaciones opuestas: que el robot se aproxime más a un paso posterior del camino que el que correspondía a su objetivo o que el robot se aleje en gran medida de su objetivo. En el primer caso, resulta mucho más útil que el robot modifique su objetivo actual por el punto del camino al que se encuentra más próximo. Por ejemplo, dado el camino *(10,20)-(20,20)-(30,20)-(40,20)*, si el robot se dirige al primer punto, pero debido a una pendiente, se desliza cuesta abajo y se sitúa en el punto *(27, 19)*, es mucho más eficiente que el nuevo objetivo sea el punto *(30,20)* pues es al que se encuentra más próximo. Por otro lado, si debido a un desvío indeseado el robot se aleja en gran medida de su objetivo, es necesario que se vuelva a calcular una nueva ruta hacia el área de interés al que el robot se dirigía. En este caso, un robot realiza esta reestructuración de la ruta si se aleja más de veinte metros de su destino.

Los dos sistemas de corrección de ruta descritos se implementan en el método *pathRevision*. Este método es invocado cada veinte iteraciones del bucle principal para comprobar si es necesario modificar la ruta actual del robot.

6.3.- Localización del Escudo Térmico

Tras realizar el análisis de las distintas rocas situadas en las áreas de interés, el robot debe localizar y aproximarse a su escudo térmico. Este objetivo requiere realizar dos tareas distintas. Por un lado, al desconocerse la situación del escudo térmico, el robot debe patrullar toda el área en el que dicho escudo puede encontrarse. Por otro, debe analizar las imágenes obtenidos a través de sus cámaras para detectar el escudo durante su patrulla. Así pues, se han desarrollado mecanismos diferentes para llevar a cabo cada una de estas tareas.

6.3.1.- Patrulla del área

Como se ha comentado anteriormente, la localización del escudo térmico es desconocida, por lo que debe rastrearse una amplia área del escenario en el que es posible encontrar dicho escudo. Para ello, tras analizar los seis yacimientos de roca, se desarrolla un itinerario a través del cual el robot debe desplazarse y realizar rastreos de las proximidades para detectar el escudo térmico.

El recorrido realizado por el robot es el siguiente:

(140,10)	(140,160)	(0,133)	(-140,133)	(-140,0)	(-140,-160)	(-110,-160)	(-110,110)	(110,110)

Figura 48.- Recorrido de la patrulla

Este recorrido transita por la zona que se muestra en esta imagen:

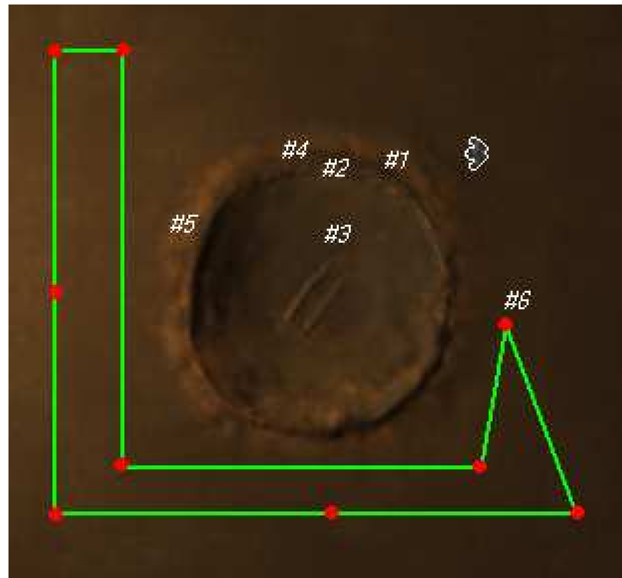

Figura 49.- Representación sobre el mapa del recorrido de patrulla

Así pues, una vez recibido este recorrido, el robot utiliza el comportamiento *Navigate* para trasladarse a cada de las localizaciones que lo componen. Sin embargo, esta vez se incluye una diferencia con respecto a la navegación entre zonas de interés. Esta diferencia consiste en que, en cada uno de los pasos intermedios por los que circula el robot hacia estos puntos de patrulla, se transita al comportamiento *HeatShield*, en el que el robot se detiene, gira su cabeza a derecha e izquierda en busca del escudo térmico y, en caso de detectarlo, varía su ruta y se dirige hacia él.

6.3.2.- Detección y aproximación al escudo

Como se introduce en el apartado anterior, cada vez que el robot realiza un paso intermedio durante su patrulla, gira su cabeza hacia ambos lados para buscar el escudo térmico. Para ello, examina las imágenes de sus cámaras panorámica y de navegación en busca de píxeles pertenecientes a dicho escudo. Los píxeles que representan al escudo se encuentran en el rango RGB delimitado por los valores (24, 32, 34) y (49, 47, 49). Cuando estos píxeles se detectan por los servicios de cámara, el robot abandona su ruta actual y se desplaza en la dirección en la que detectó el escudo, con el fin de alcanzar su posición.

El comportamiento en el que el robot rastrea ambos laterales en busca del escudo se denomina *HeatShield* y el estado al que transita cuando lo identifica y se dirige hacia él se conoce como *HeatFound*.

6.4.- Condiciones de finalización de la misión

La misión finaliza cuando se transita al comportamiento EndMission, en el que se simula el envío de un mensaje de término de misión a la Tierra a través del servicio que controla el puerto de comunicaciones. La misión se da por terminada si se han cumplido todos los objetivos de la misma, o si es imposible cumplir ningún objetivo más. El primer caso se cumple cuando se han analizado los seis yacimientos de roca, así como detectado y localizado el escudo térmico. Para conocer el cumplimiento de estos hechos se utilizan las variables lógicas *rocksAnalyzed* y *shieldLocated*, respectivamente. Al principio de cada iteración del bucle principal se comprueba si ambas variables tienen valor "verdadero", lo que indicaría que se han cumplido todos los objetivos, y, en este caso, transita al estado *EndMission*.

En lo que respecta a las condiciones de fracaso de misión, éstas se corresponden con el vuelco del robot, es decir, si las ruedas del robot pierden completamente el contacto con el suelo la misión se considera abortada, pues éste no posee ningún mecanismo para recuperar su posición normal.

6.5.- Diagrama de estados: transición entre comportamientos.

Como se muestra en los anteriores capítulos, la lógica funcional del robot consta de un bucle en el que, en cada iteración, el robot muestra un determinado comportamiento en base al estado interno del servicio que lo controla.

Para aportar mayor claridad al comportamiento general del robot, se incluye la siguiente imagen en la que se representa la máquina de estados que determina dicho comportamiento:

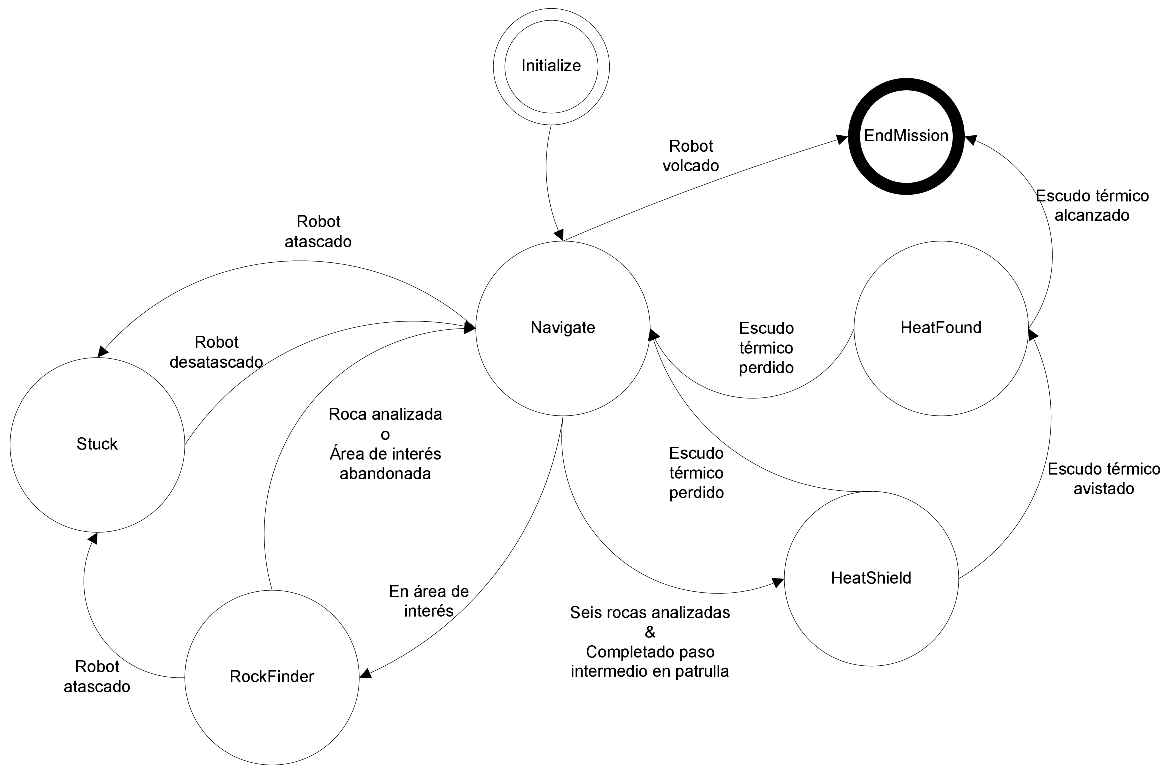

Figura 50.- Diagrama de transiciones entre comportamientos

6.6- Resultados

Tras implementar la serie de comportamientos descritos en el apartado anterior, se ejecutó una prueba de validación a través de la interfaz del desafío. Dicha prueba de validación consistía en la ejecución completa del comportamiento del robot y el envío automático de los resultados obtenidos al servidor de la competición. Esta prueba se realizó el día 19 de mayo a la hora 01:00 y la puntuación obtenida fue de 725 puntos, es decir, la máxima posible. Esta puntuación quedó almacenada en el servidor de la competición.

El día 27 de mayo el coordinador jefe de la competición, Rogerio Panigassi, envió una notificación vía correo electrónico en la que se anunciaba que la elevada puntuación obtenida reunía posibilidades para que su autor fuese seleccionado para poder participar en la final celebrada en El Cairo. Además, y dado que en ese momento el equipo de jueces sólo contaba con los resultados, pero nada de código, se requería el envío del código fuente utilizado para competir. Este código fue remitido a la dirección de correo electrónico utilizada como contacto, incluyendo los comentarios en inglés, pues es el idioma oficial de la competición, y el día 4 de junio se anunció la lista de clasificados. En esta lista se mostraba la posición obtenida por cada uno de los participantes ordenados, en primer lugar, por la puntuación obtenida en la misión y, en segundo lugar, por una puntuación asignada por los miembros del jurado en base a unos requisitos internos que no fueron publicados.

En esta ocasión, se presentaron cincuenta y seis participantes, consiguiendo alcanzar la máxima puntuación cuarenta y dos de ellos. Entre estos cuarenta y dos participantes, el código realizado durante este proyecto, y descrito anteriormente, ocupa la decimosegunda posición. Por tanto, no permitió el avance a la ronda final de la competición. El nombre de usuario utilizado para participar fue "*Avalor*", y en la siguiente imagen, que muestra las doce primeras posiciones de la clasificación, puede comprobarse que ocupó la duodécima posición:

Rank	Country/Region	User name	Score	Advanced
1	LK	Aravinda DP	725.000	●
1	CA	byronknoll	725.000	●
1	AR	gauna	725.000	●
1	CZ	Imro	725.000	●
1	CZ	Lucaso	725.000	●
1	CN	zero.lin	725.000	●
1	DE	_andi	725.000	
1	CN	aldose *	725.000	
1	CN	alex890714 *	725.000	
1	DE	andi_ *	725.000	
1	IN	anirbanbob *	725.000	
1	ES	Avalor *	725.000	

Figura 51.- Clasificación de la segunda ronda

Capítulo 7:

Conclusiones y Líneas Futuras

La elaboración del presente proyecto, ha requerido un intenso proceso de análisis de las características y posibilidades que ofrece la herramienta Microsoft Robotics Developer Studio (MRDS). Durante este análisis se ha puesto de manifiesto las enormes posibilidades que esta herramienta aporta al futuro de la programación robótica.

En primera instancia, resulta destacable el fomento del interés en la robótica que esta herramienta promueve, pues aporta una versión académica gratuita y completamente funcional para estudiantes e investigadores. Por otro lado, también posee gran relevancia la transparencia hacia el programador con la que se realizan las tareas de coordinación y comunicación entre servicios, pues de manera sencilla es posible incluir directivas, como las comentadas en los capítulos 3 y 5, que permiten manejar la concurrencia entre los distintos procesos o servicios. Estas directivas forman parte de los entornos de ejecución Decentralized Software Services (DSS), en el que se define la aplicación como un conjunto de servicios independientes que se comunican entre sí para aportar la funcionalidad global de dicha aplicación, y Concurrency and Coordination Runtime (CCR), que incluye los mecanismos necesarios para realizar la comunicación entre los distintos servicios que conforman una aplicación y la coordinación entre las tareas realizadas en los mismo. Así pues, a través de estas conclusiones sobre la herramienta MRDS queda patente la consecución del primero de los objetivos, pues en el capítulo 3 se ha realizado una descripción de las características más relevantes de esta herramienta poniendo de manifiesto aquellas cuyo conocimiento resulta imprescindible para cualquier programador que desee utilizar dicha herramienta. Por otro lado, se puede comprobar que se han obtenido las destrezas necesarias para comprender y utilizar la metodología de programación robótica MRDS, ya que se ha hecho uso de ella para realizar una solución robótica en la que se hace frente a los retos más actuales de la navegación robótica en entorno desconocido, a través de una competición de robótica fundamentada en una misión real acontecida en Marte.

En lo que respecta al segundo objetivo, también puede afirmarse que ha sido conseguido, pues, como se muestra en el apartado 6, se ha elaborado una solución robótica capaz de enfrentarse a algunos de los retos actuales de la navegación robótica en entorno desconocido. Entre las distintas técnicas implementadas, se han incluido mecanismos para planificar una ruta de navegación global entre dos puntos del espacio, utilizando como único soporte a la localización un sistema de posicionamiento global GPS y las coordenadas de Euler que definen la orientación del robot en su posición. Además, durante el desplazamiento del robot, se han encontrado diversos factores adversos que dificultan el proceso de navegación global, como las pendientes pronunciadas que imposibilitan el tránsito del robot, o desviaban su trayectoria, o bloqueaban la rotación de sus ruedas.

Frente a los elementos que entorpecían la navegación global, se diseñaron mecanismos de navegación local con el fin de evitar su efecto adverso o, al menos, minimizar sus consecuencias y conseguir retornar al camino global planificado. Dichos mecanismos se basan, fundamentalmente, en la prealimentación, es decir, en la previsión de futuras contrariedades derivadas de un conocimiento que se posee, como es el caso del abandono del cráter en el que el robot debía internarse. En este caso, el robo era consciente de que se encontraba en su interior, por lo que, en previsión de su difícil salida, elaboraba un camino diseñado para evitar las grandes pendientes que era posible encontrar. Por otro lado, también se incluyen mecanismos de control de la frenada, pues frenadas bruscas pueden ocasionar el vuelco del robot si éste se encuentra sobre un terreno con una determinada pendiente. En cuanto a los obstáculos fijos que se encuentran en el terreno, se incluyen mecanismos de detección de atasco, basados en la monitorización del desplazamiento del robot para descubrir situaciones en las que éste no se traslada a pesar de que sus ruedas están en movimiento, que indican al robot que debe abandonar temporalmente su ruta planificada para efectuar determinadas acciones que lo liberen de su situación de atasco y poder retomar la ruta inicial planificada. Además se han incluido mecanismos para corregir desvíos en la ruta ocasionados por deslizamientos en pendientes o evasión de obstáculos, mediante los cuales es posible modificar el camino planificado o, en su caso, elaborar uno nuevo.

Con respecto a la detección de elementos del entorno, se han incluido procedimientos de análisis de imagen con la finalidad de identificar elementos relevantes, que en este caso han sido muestras de roca y un escudo térmico. En estos procedimientos se han analizado los píxeles de los que se componen las imágenes obtenidas a través de las cámaras con las que cuenta el robot para localizar aquellos que, por el color que representan, forman parte de los elementos a identificar. La detección de dichos objetos, así como la localización relativa de los mismos con respecto al robot, mediante técnicas de posicionamiento relativo, han sido utilizadas para modificar el comportamiento del robot con el fin de interactuar con los mismos, como en el caso del proceso de análisis de roca.

Finalmente, también puede resaltarse el esfuerzo realizado por simplificar los cálculos necesarios para llevar a cabo las tareas que componen los anteriores procedimientos. El objetivo de esta simplificación ha sido optimizar el tiempo de respuesta del robot y ha consistido, fundamentalmente, en la sustitución de operaciones complejas como las que implican números decimales, o en coma flotante, por otras más sencillas, como ocurre en el caso del posicionamiento relativo de elementos del entorno a través de las cámaras. En este proceso se ha disminuido en pequeña medida la precisión del posicionamiento a favor de una gran disminución de la complejidad computacional, a través de la sustitución de operaciones decimales por números aproximados conocidos más simples.

En cuanto a las soluciones aportadas para cada uno de estos retos de navegación robótica, podemos afirmar que han sido eficaces, pues el robot ha sido capaz de enfrentarse a dichos retos y superarlos satisfactoriamente. Este hecho puede ser constatado utilizando los resultados de la competición, pues se consiguió la máxima puntuación obtenible en el desafío planteado y se consiguió un duodécimo puesto, entre los cuarenta y dos participantes que también consiguieron la puntuación máxima, en la clasificación general de la competición internacional Imagine Cup '09. Así pues, los objetivos marcados al principio del proyecto han sido superados, por lo que puede afirmarse que los resultados obtenidos tras la realización del proyecto son satisfactorios.

En lo que respecta a futuras líneas de desarrollo, aunque haya terminado la competición por este año, no son inexistentes, pues es posible realizar una investigación más profunda de los problemas más difíciles a los que se ha hecho frente, como, por ejemplo, la salida del cráter, para evitar recurrir a indicar manualmente un punto de salida al mismo. Estos conocimientos estarán encaminados a preparar la participación en una nueva edición de la competición Imagine Cup en el año 2010 y posteriores. Además, en estas nuevas ediciones se contará con mayores posibilidades de éxito, pues, paralelamente a la implementación de la solución, no será necesario efectuar un proceso de adquisición de conocimientos de la programación robótica MRDS, como ha ocurrido en la presente edición, sino que éstos ya habrán sido obtenidos.

Capítulo 8:

Bibliografía y Referencias

Anónimo. “*La investigación del cráter Endurance: profundizando en el pasado acuoso de Marte”*. http://www.astroenlazador.com/article.php3?id_article=315

Borenstein, J., Everett, H.R., Feng, L., Wehe, D.,“*Mobile Robot Positioning - Sensors and Techniques*”. Journal of Robotic Systems. 1997

Crowley, **J**.**L**., **Demazeau**, **Y**., “*Principles and Techniques for Sensor Data Fusion*”. Grenoble Cedex, Francia, 1993.

Fox, **D**., **Thrun**, **S**., **Burgard**, **W**., “*Markov Localization for Mobile Robots in Dynamic Environments*”. *Journal of Artificial Intelligence Research*. 1999.

Gallardo López, **D**., “*Aplicación del Muestreo Bayesiano en Robots Móviles: Estrategias para Localización y Estimación de Mapas del Entorno*”. Tesis Doctoral. 1999

García Barrales, C. I. , *“Monte Carlo aplicado a la auto-localización de robots”*. Tesis doctoral. Universidad de las Américas Puebla, México. 2005

Handschin, **J**., “*Monte Carlo techniques for prediction and filtering of non-linear stochastic processes*”, Automatica, 1970.

Hwang, **Y**.**K**., **Ahuja**, **N**., “*A Potential Field Approach to Path Planning*” para *IEEE Transactions on Robotics and Automation*. Universidad de Illinois, 1992.

Jet Propulsion Laboratory, “*NASA Mars Rover Opportunity Gets Green Light to Enter Endurance Crater”*. http://www.spaceref.com/news/viewpr.html?pid=14353

Latombe, **J**.**C**., “*Robot Motion Planning*”. Norwood, Massachusetts, Kluwer Academic Publishers, 1991.

Laubach, **S**.**L**., **Burdick**, **J**.**W**., “*An Autonomous Sensor-Based Path-Planner for planetary Microrovers”, Instituto Tecnológico de California*. Pasadena, California.

Lazanas, **A**., **Latombe**, **J**.**C**., "*Landmark-Based Robot Navigation*". *Procedimientos de la Décima Conferencia Anual en Inteligencia Artificial*. San Jose, California, Julio de 1992

Leonard, **J.J.**, **Durrant-Whyte**, **H**. **F.**, "*Mobile robot localization by tracking geometric beacons*" para *IEEE Transactions on Robotics and Automation*, 1991.

López, **E.**, **Barea**, **R**., **Bergasa**, **L.M.**, **Escudero**, **M.S.**, "*Aplicación del método de Markov a la localizacióndе un robot móvil en entornos interiores"*. 2002

Malik, **T**., "*At Endurance Crater, Opportunity Rover Treads Carefully*".
http://www.space.com/missionlaunches/rovers_update_040506.html

Moravec, **H**., **Elfes**, **A**.**E**., "*High Resolution Maps from Wide Angle Sonar*", para *Procedimientos de la Conferencia Internacional IEEE sobre Robótica y Automatización*. IEEE Press, Saint Louis, Missouri, 1985.

Negenborn, **R**., "*Robot Localization and Kalman Filters : On finding your position in a noisy world*". Tesis doctoral en la Universidad de Utrecht. 2003.

Riisgard, **S**., **Blas**, **M**.**R**., "*SLAM for Dummies*".
http://ocw.mit.edu/NR/rdonlyres/Aeronautics-and-Astronautics/16-412JSpring-2005/9D8DB59F-24EC-4B75-BA7A-F0916BAB2440/0/1aslam_blas_repo.pdf

Sánchez, **A**., **Sanz-Bobi**, **M**.**A**., "*Global path planning in Gaussian probabilistic maps*" *Journal of Intelligent and Robotics Systems*. 2004

Siegwart, **R**., **Nourbakhsh**, **I**.**R**., "*Introduction to Autonomous Mobile Robots*". Editorial MIT Press, 2004.

Thrun, **S**., **Burgard**, **W**., **Fox**, **D**., "*A Probabilistic Approach to Concurrent Mapping and Localization for Mobile Robots*". Kluwer Academic Publishers, 1998.

Apéndice A:
Instalación del entorno

En este proyecto se muestra cómo elaborar una aplicación robótica utilizando la tecnología MRDS. Para ello, es necesario adquirir las herramientas necesarias para programar dicha aplicación. Como ya se ha indicado en apartados anteriores, las herramientas necesarias son:

- **Microsoft Visual Studio 2008**

Esta herramienta se puede adquirir a través del portal Microsoft Developer Network Academia Alliance (MSDNAA), si somos usuarios registrados. Para estudiantes de la Universidad Carlos III Madrid el enlace web es http://msdn30.e-academy.com/elms/Storefront/Home.aspx?campus=ucarlos_info.

Para el resto de usuarios, Microsoft proporciona una versión express de esta herramienta en el sitio http://www.microsoft.com/express/download/. Deberemos descargar la versión correspondiente al lenguaje C#.

- **Microsoft Robotics Developer Studio 2008**

Igual que ocurre con la herramienta anterior, es posible descargar una versión académica de la misma a través de la página de MSDNAA.

Sin embargo, también es posible descargar una versión express gratuita en la dirección:

http://www.microsoft.com/downloads/details.aspx?FamilyID=84c5b49f-0f9c-4182-a267-a951328d3fbd&displaylang=en.

Finalmente, es necesario descargar el paquete que contiene el proyecto Visual Studio sobre el que trabajaremos. Las diferentes descargas se encuentran en www.robochamps.com en el apartado "Downloads\Challenges". Como muestra la siguiente imagen, en este proyecto trabajaremos sobre el desafío *ImagineCup MarsChallenge.*

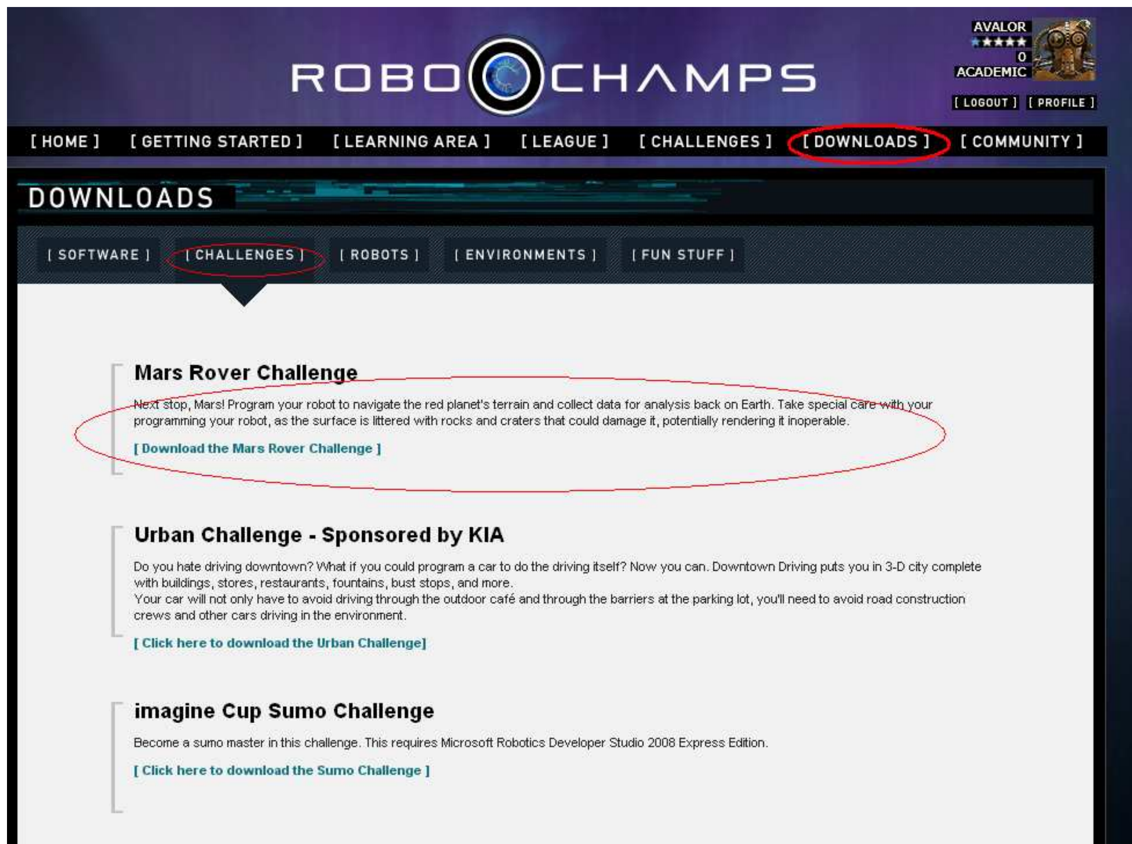

Figura 52.- Pantalla de descarga

Si accedemos al enlace de descarga, obtenemos el archivo de instalación setup.exe. Debemos ejecutarlo y seleccionar como directorio de instalación el mismo en el que se encuentre instalado MRDS:

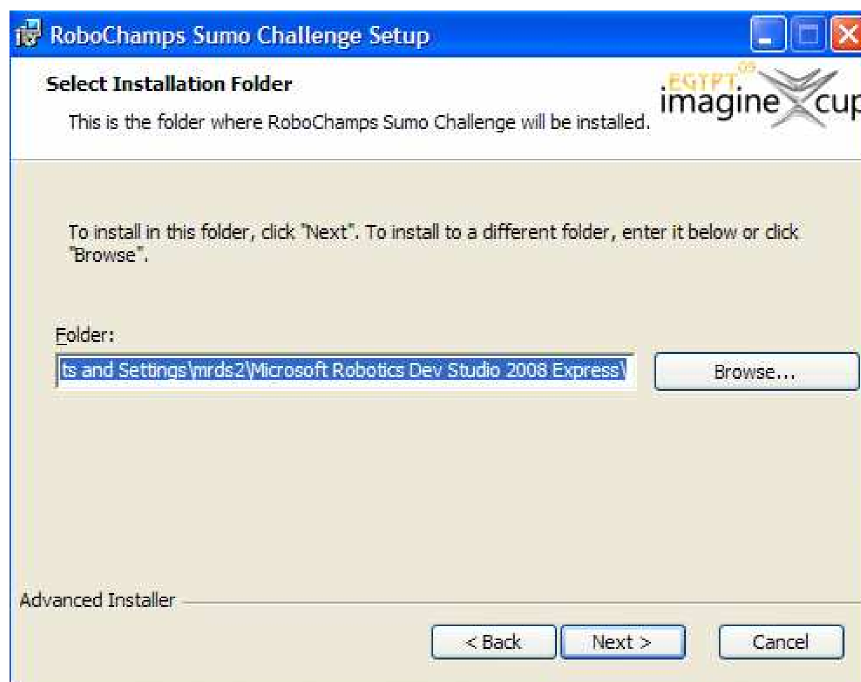

Figura 53.- Ventana de instalación

Tras realizar los anteriores pasos con éxito, el entorno de trabajo queda completamente instalado. Para poder comenzar a desarrollar la solución, debemos acceder a la carpeta “Robochamps\Mars\Sample” que se encuentra dentro del directorio de instalación de MRDS.

El contenido de esta carpeta debe ser el siguiente:

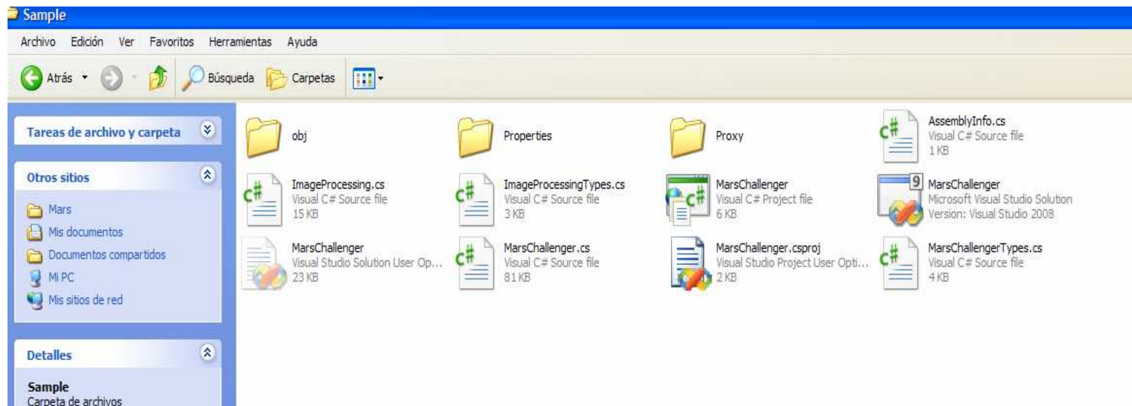

Figura 54.- Carpeta "RoboChamps\Mars\Sample”

Como podemos comprobar, se ha creado una serie de archivos que forman parte del proyecto Visual Studio sobre el que comenzaremos a trabajar. Para acceder al código del proyecto, debemos abrir el archivo “MarsChallenger.sln” y se nos mostrará una pantalla como la siguiente:

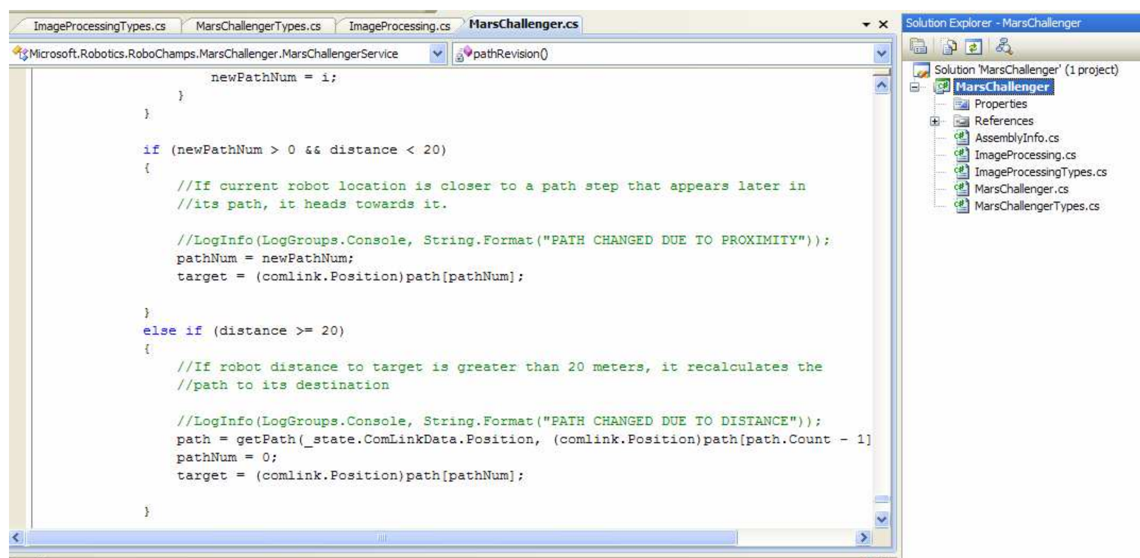

Figura 55.- Interfaz del proyecto

Una vez se hayan instalado los ficheros, sólo resta sustituir el código inicial por el incluido en el apéndice D, en sus ficheros correspondientes.

Apéndice B:
Contenido del CD

Adjunto a la presente memoria, se incluye un CD con la siguiente estructura y contenido:

- **Carpeta "Código"**: contiene los ficheros necesarios para instalar el desafío Mars Challenge y la solución desarrollada durante el presente proyecto. Estos ficheros son:

    - "Mars.rar": contiene el código fuente de la solución implementada.

    - "setup.exe": instalador del desafío. Contiene todas las librerías y servicios necesarios para ejecutar la aplicación de ejemplo. Ejecutando este fichero, no es preciso descargarlo de la página www.Robochamps.com.

- **Carpeta "Memoria":** contiene la presente memoria en formato PDF.

Apéndice C:
Planificación

Las distintas tareas que han sido realizadas para elaborar el siguiente proyecto pueden resumirse de la siguiente manera:

- **1.- Análisis MRDS:** investigar la tecnología MRDS para comprender los elementos que la componen y adquirir conocimiento sobre sus posibilidades y metodología de trabajo.

- **2.- Documentación sobre navegación robótica:** análisis de la situación actual de la navegación robótica. Dicho análisis se realizó sobre los distintos problemas que componen la navegación robótica y las soluciones existentes a los mismos.

- **3.- Implementación de la solucion:** se compone de varias subtares.

    - Navegación Global: implementar un mecanismo mediante el cual el robot sea capaz de desplazarse entre dos puntos del espacio.

    - Navegación Local: incluir procedimientos de soporte a la navegación global que permitan evitar obstáculos del entorno. Además, se incluyen tareas de identificación de elementos de interés, en este caso, las muestras de roca.

    - Detección del escudo térmico: implementar un procedimiento que permita al robot detectar su escudo térmico y acercarse a él.

    - Optimización de código: modificar los procedimientos y técnicas implementados con la finalidad de reducir la complejidad computacional de la solución.

- **Preparación de la memoria:** elaboración del presente documento.

A cada una de estas tareas se le asignó un período de tiempo durante el cual debían ser ejecutadas en su totalidad. En la siguiente imagen, realizada utilizando la herramiento de planificación Microsoft Project, se muestra dicha asignación:

		Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
1	✓	Análisis MRDS	31 días	dom 01/02/09	vie 13/03/09	
2	✓	Documentación del estado del arte de la navegación robótica	48 días	lun 16/03/09	mié 20/05/09	
3	✓	**Implementación de la solución**	**22 días**	**lun 20/04/09**	**mar 19/05/09**	
4	✓	Navegación Global	10 días	lun 20/04/09	vie 01/05/09	
5	✓	Navegación Local	10 días	lun 04/05/09	vie 15/05/09	
6	✓	Detección del escudo térmico	6 días	lun 11/05/09	lun 18/05/09	
7	✓	Optimización de código	7 días	lun 11/05/09	mar 19/05/09	
8		Preparación de la memoria	28 días	mié 20/05/09	lun 29/06/09	3

Figura 56.- Asignacion temporal de tareas

En la siguiente imagen se muestra el seguimiento realizado de cada una de las tareas en su fecha de finalización:

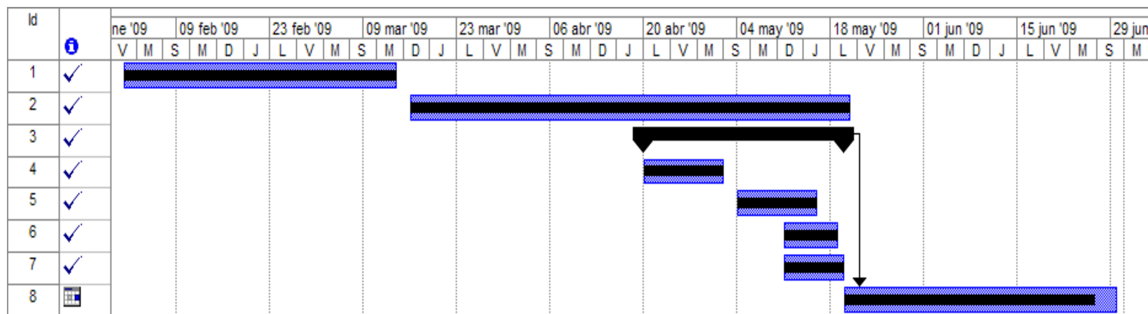

Figura 57.- Seguimiento de tareas

Como podemos observar, todas las tareas se realizaron en el período de tiempo previsto, excepto la elaboración del presente documento. La fecha de terminación del documento de memoria estaba prevista para el día 29 de junio, pero se retrasó hasta el día 1 de julio por problemas de sobrecarga de trabajo derivados de a una planificación inicial demasiado optimista.

Apéndice D:
Código Implementado

Fichero ImageProcessingTypes.cs

```
using System;
using Microsoft.Ccr.Core;
using System.Collections.Generic;
using System.ComponentModel;
using Microsoft.Dss.ServiceModel.DsspServiceBase;
using Microsoft.Dss.Core.Attributes;
using Microsoft.Dss.ServiceModel.Dssp;
using W3C.Soap;

using submgr = Microsoft.Dss.Services.SubscriptionManager;
using webcam = Microsoft.Robotics.Services.WebCam.Proxy;

namespace Microsoft.Robotics.RoboChamps.MarsChallenger.ImageProcessing
{
    public sealed class Contract
    {
        [DataMember]
        public const String Identifier =
"http://www.robochamps.com/2009/02/marschallenger/imageprocessing.html
";
    }

    [DataContract]
    public class ImageProcessingState
    {
        [DataMember]
        public DateTime LastProcessing { get; set; }
    }

    /// <summary>
    /// Class that contains result for image processing
    /// </summary>
    [DataContract]
    public class ImageResult
    {
        [DataMember]
        public DateTime Timestamp { get; set; }

        [DataMember]
        public int XMean { get; set; }

        [DataMember]
        public int YMean{ get; set; }

        [DataMember]
        public int ShieldArea { get; set; }
```

```
        [DataMember]
        public int RockPosition { get; set; }

        [DataMember]
        public int RockArea { get; set; }


        // You could add what you want in image result
    }


    [ServicePort]
    public class ImageProcessingOperations :
PortSet<DsspDefaultLookup, DsspDefaultDrop, Subscribe, ProcessImage>
    {
        public virtual PortSet<SubscribeResponseType, Fault>
Subscribe(IPort notificationPort, params Type[] types)
        {
            SubscribeRequestType body = new SubscribeRequestType();
            Subscribe operation = new Subscribe(body);
            operation.NotificationPort = notificationPort;
            if ((types != null))
            {
                body.TypeFilter = new string[types.Length];
                for (int index = 0; (index < types.Length); index =
(index + 1))
                {
                    body.TypeFilter[index] =
DsspServiceBase.GetTypeFilterDescription(types[index]);
                }
            }
            this.Post(operation);
            return operation.ResponsePort;
        }

    }

    public class Subscribe : Subscribe<SubscribeRequestType,
PortSet<SubscribeResponseType, Fault>>
    {
        public Subscribe() : base() { }
        public Subscribe(SubscribeRequestType body) : base(body) { }
        public Subscribe(SubscribeRequestType body,
PortSet<SubscribeResponseType, Fault> responsePort) : base(body,
responsePort) { }
    }

    public class ProcessImage : Update<ImageResult,
PortSet<DefaultUpdateResponseType, Fault>>
    {
        public ProcessImage() : base() { }
        public ProcessImage(ImageResult body) : base(body) { }
        public ProcessImage(ImageResult body,
PortSet<DefaultUpdateResponseType, Fault> responsePort) : base(body,
responsePort) { }
    }
}
```

Fichero ImageProcessing.cs

```
using System;
using Microsoft.Ccr.Core;
using System.Collections.Generic;
using System.ComponentModel;
using Microsoft.Dss.ServiceModel.DsspServiceBase;
using Microsoft.Dss.Core.Attributes;
using Microsoft.Dss.ServiceModel.Dssp;
using W3C.Soap;

using submgr = Microsoft.Dss.Services.SubscriptionManager;
using webcam = Microsoft.Robotics.Services.WebCam.Proxy;

namespace Microsoft.Robotics.RoboChamps.MarsChallenger.ImageProcessing
{
    /// <summary>
    /// Processes webcam image and sends notifications with result
    /// </summary>
    [DisplayName("Mars Image Processor")]
    [Description("The Mars Challenger Sample Image Processing
Service")]
    [Contract(Contract.Identifier)]
    public class ImageProcessingService : DsspServiceBase
    {
        [ServiceState]
        private ImageProcessingState _state = new
ImageProcessingState();

        [ServicePort("/ImageProcessingService", AllowMultipleInstances
= true)]
        private ImageProcessingOperations _mainPort = new
ImageProcessingOperations();

        #region Partners

        [Partner("Camera", Contract = webcam.Contract.Identifier,
CreationPolicy = PartnerCreationPolicy.UsePartnerListEntry)]
        private webcam.WebCamOperations _camPort = new
webcam.WebCamOperations();
        private webcam.WebCamOperations _camNotify = new
webcam.WebCamOperations();

        [SubscriptionManagerPartner]
        private submgr.SubscriptionManagerPort _submgrPort = new
submgr.SubscriptionManagerPort();

        #endregion

        /// <summary>
        /// Dictionary that hold image processor, depending service
name
        /// </summary>
        private Dictionary<string, IImageProcessor> _processors =
            new Dictionary<string, IImageProcessor>(new
ProcessorDictionaryComparer());

        /// <summary>
        /// Current service processor
        /// </summary>
```

```
        private IImageProcessor _processor = null;



        /// <summary>
        /// Initializes the service.
        /// Associates service name and image processor.
        /// </summary>
        /// <param name="creationPort"></param>
        public ImageProcessingService(DsspServiceCreationPort
creationPort) :
            base(creationPort)
        {
            _processors.Add("pancamprocessor", new PanCamProcessor());
            _processors.Add("navcamprocessor", new NavCamProcessor());
            _processors.Add("frontcamprocessor", new
BasicCamProcessor());
            _processors.Add("rearcamprocessor", new
BasicCamProcessor());
        }

        /// <summary>
        /// Service Start
        /// </summary>
        protected override void Start()
        {
            base.Start();

            // Gets the processor associated to this service
            _processors.TryGetValue(ServiceInfo.Service, out
_processor);

            // Initialization
            _state.LastProcessing = DateTime.MinValue;

            // Subscribe to webcam notification
            MainPortInterleave.CombineWith(Arbiter.Interleave(
                new TeardownReceiverGroup(),
                new ExclusiveReceiverGroup
                (

Arbiter.ReceiveWithIterator<webcam.UpdateFrame>(true, _camNotify,
UpdateFrameHandler)
                ),
                new ConcurrentReceiverGroup()));
            _camPort.Subscribe(_camNotify,
typeof(webcam.UpdateFrame));
        }

        /// <summary>
        /// Handler for Subscribe message
        /// </summary>
        [ServiceHandler(ServiceHandlerBehavior.Concurrent)]
        public IEnumerator<ITask> SubscribeHandler(Subscribe
subscribe)
        {
            yield return Arbiter.Choice(
                SubscribeHelper(_submgrPort, subscribe,
subscribe.ResponsePort),
                (success) => { },
                (error) => LogError(error));
```

```
        }

        /// <summary>
        /// Handler for ProcessImage message.
        /// It's called internally and just sends notifications.
        /// </summary>
        [ServiceHandler(ServiceHandlerBehavior.Exclusive)]
        public IEnumerator<ITask> ProcessImageHandler(ProcessImage
process)
        {

process.ResponsePort.Post(DefaultUpdateResponseType.Instance);
            SendNotification<ProcessImage>(_submgrPort, process);
            yield break;
        }

        /// <summary>
        /// Handler for webcam notifications
        /// </summary>
        protected IEnumerator<ITask>
UpdateFrameHandler(webcam.UpdateFrame update)
        {
            // Discards old frame
            if (update.Body.TimeStamp.CompareTo(_state.LastProcessing)
< 0)
            {
                yield break;
            }

            _state.LastProcessing = update.Body.TimeStamp;

            // Queries a new frame
            webcam.QueryFrameResponse queryResponse = null;
            yield return Arbiter.Choice(_camPort.QueryFrame(),
(success) => queryResponse = success, LogError);

            if (_processor != null && queryResponse != null)
            {
                // Calls the image processor and post the result on
the main port (to send notifications)
                _mainPort.Post(new
ProcessImage(_processor.Process(queryResponse)));
            }
        }
    }

    #region Helpers

    /// <summary>
    /// Equality comparer for processor dictionary.
    /// Compares two URI with the final part
    /// </summary>
    internal class ProcessorDictionaryComparer :
IEqualityComparer<string>
    {
        #region IEqualityComparer<string> Members

        public bool Equals(string x, string y)
        {
            return Normalize(x) == Normalize(y);
```

```
        }

        public int GetHashCode(string obj)
        {
            return Normalize(obj).GetHashCode();
        }

        #endregion

        private string Normalize(string s)
        {
            if (s == null)
            {
                return s;
            }
            int i = s.LastIndexOf('/');
            if (i != -1)
            {
                s = s.Substring(i + 1);
            }
            return s;
        }
    }

    #endregion

    #region Processors

    /// <summary>
    /// Processor interface
    /// </summary>
    internal interface IImageProcessor
    {
        /// <summary>
        /// Returns an image result after analyzing a query frame
response
        /// </summary>
        ImageResult Process(webcam.QueryFrameResponse image);



    }

    /// <summary>
    /// Processor for panoramic camera
    /// </summary>
    internal class PanCamProcessor : IImageProcessor
    {


        public ImageResult
Process(Microsoft.Robotics.Services.WebCam.Proxy.QueryFrameResponse
image)
        {
            int offset = 0;
            int xMean = 0;
            int yMean = 0;
            int area = 0;

            for (int y = 0; y < image.Size.Height; ++y)
            {
```

```
                offset = y * image.Size.Width * 3;

                for (int x = 0; x < image.Size.Width; ++x)
                {
                    int r, g, b;

                    b = image.Frame[offset++];
                    g = image.Frame[offset++];
                    r = image.Frame[offset++];


                    if ((r > 34 && r < 49) && (g > 32 && g < 47) && (b
> 34 && b < 49))
                    {
                        //Shield spotted
                        xMean += x;
                        yMean += y;
                        area++;
                    }
                }
            }

            ImageResult result = new ImageResult();
            if (area > 0)
            {
                xMean /= area;
                yMean /= area;
            }
            result.ShieldArea = area;
            result.XMean = xMean;
            result.YMean = yMean;
            result.Timestamp = DateTime.Now;
            return result;
        }
    }

    /// <summary>
    /// Processor for navigation camera
    /// </summary>
    internal class NavCamProcessor : IImageProcessor
    {


        public ImageResult
Process(Microsoft.Robotics.Services.WebCam.Proxy.QueryFrameResponse
image)
        {
            int offset = 0;//Shield
            int xMean = 0;//Shield
            int yMean = 0;//Shield
            int shieldArea = 0;//Shield
            int rockArea = 0; //Rocks


            //Frame divided twice: 5 Vertontal and 5 Horizical
rectangles

            //Number of pixels of the shield that are seen in a
certain Vertical portion
            int Horiz1 = 0;
            int Horiz2 = 0;
```

```
            int Horiz3 = 0;
            int Horiz4 = 0;
            int Horiz5 = 0;

            //Number of pixels of the shield that are seen in a
certain Horizontal portion
            int Vert1 = 0;
            int Vert2 = 0;
            int Vert3 = 0;
            int Vert4 = 0;
            int Vert5 = 0;



            ImageResult result = new ImageResult();

            for (int y = 0; y < image.Size.Height; ++y)
            {
                offset = y * image.Size.Width * 3;

                for (int x = 0; x < image.Size.Width; ++x)
                {
                    int r, g, b;

                    b = image.Frame[offset++];
                    g = image.Frame[offset++];
                    r = image.Frame[offset++];

                    if ((r > 21 && r < 32) && (g > 11 && g < 19) && (b
> 4 && b < 10))
                    {
                        //Rock spotted
                        rockArea++;
                        if (x <= image.Size.Width / 5)
                        {
                            Vert1++;
                        }
                        else if (x <= ((2 * image.Size.Width) / 5) +
5)
                        {
                            Vert2++;
                        }
                        else if (x <= (((3 * image.Size.Width) / 5) -
15))
                        {
                            Vert3++;
                        }
                        else if (x <= (4 * image.Size.Width) / 5)
                        {
                            Vert4++;
                        }
                        else
                        {
                            Vert5++;
                        }

                        if (y <= image.Size.Height / 5)
                        {
                            Horiz1++;
                        }
                        else if (y <= (2 * image.Size.Height) / 5)
```

```
                    {
                        Horiz2++;
                    }
                    else if (y <= (3 * image.Size.Height) / 5)
                    {
                        Horiz3++;
                    }
                    else if (y <= (4 * image.Size.Height) / 5)
                    {
                        Horiz4++;
                    }
                    else
                    {
                        Horiz5++;
                    }
                }

                if ((r > 34 && r < 49) && (g > 32 && g < 47) && (b
> 34 && b < 49))
                {
                    //Shield spotted
                    xMean += x;
                    yMean += y;
                    shieldArea++;
                }
            }
        }

        int res = 0;

        if (Horiz5 > 2)
        {
            if (Vert3 > 2)
            {
                res = 3;
            }
            else if (Vert5 > 2 || Vert4 > 2)
            {
                res = 5;
            }
            else if (Vert1 > 2 || Vert2 > 2)
            {
                res = 1;
            }
        }
        else if (Vert3 > 2)
        {
            res = 3;
        }
        else if (Vert1 > 2 || Vert2 > 2)
        {
            res = 2;
        }
        else if (Vert4 > 2 || Vert5 > 2)
        {
            res = 4;
        }
        else
        {
            res = 0;
        }
```

```
            if (shieldArea > 0)
            {
                xMean /= shieldArea;
                yMean /= shieldArea;
            }
            else
            {
                xMean = 0;
                yMean = 0;
            }

            result.RockPosition = res;
            result.ShieldArea = shieldArea;
            result.XMean = xMean;
            result.YMean = yMean;
            result.RockArea = rockArea;

            result.Timestamp = DateTime.Now;
            return result;
        }

    }

    /// <summary>
    /// Processor for rear and front cameras
    /// </summary>
    internal class BasicCamProcessor : IImageProcessor
    {
        public ImageResult
Process(Microsoft.Robotics.Services.WebCam.Proxy.QueryFrameResponse
image)
        {
            int offset = 0;

            int xMean = 0;
            int yMean = 0;
            int area = 0;




            for (int y = 0; y < image.Size.Height; ++y)
            {
                offset = y * image.Size.Width * 3;

                for (int x = 0; x < image.Size.Width; ++x)
                {
                    int r, g, b;

                    b = image.Frame[offset++];
                    g = image.Frame[offset++];
                    r = image.Frame[offset++];

                    int sum = (r + g + b);
                    if (sum>=60&&sum<=80)
```

```
                    {
                        //Shadow spotted
                        xMean += x;
                        yMean += y;
                        area++;

                    }
                }
            }

            ImageResult result = new ImageResult();
            if (area > 0)
            {
                xMean /= area;
                yMean /= area;
            }
            result.Timestamp = DateTime.Now;
            result.ShieldArea = area;
            result.XMean = xMean;
            result.YMean = yMean;
            return result;
        }
    }

    #endregion
}
```

Fichero MarsChallengerTypes.cs

```
using Microsoft.Ccr.Core;
using Microsoft.Dss.Core.Attributes;
using Microsoft.Dss.ServiceModel.Dssp;
using System;
using System.Collections.Generic;
using W3C.Soap;
using Microsoft.Dss.Core.DsspHttp;

using drive = Microsoft.Robotics.Services.Drive.Proxy;
using processor =
Microsoft.Robotics.RoboChamps.MarsChallenger.ImageProcessing;
using comlink = Microsoft.Robotics.RoboChamps.MarsComLink.Proxy;
using spectro = Microsoft.Robotics.RoboChamps.MarsSpectrometer.Proxy;
using pantilt = SimplySim.Robotics.PanTilt.Proxy;
using arm = Microsoft.Robotics.Services.ArticulatedArm.Proxy;

namespace Microsoft.Robotics.RoboChamps.MarsChallenger
{
    public sealed class Contract
    {
        [DataMember]
        public const String Identifier =
"http://www.robochamps.com/2009/02/marschallenger.html";
    }

    [DataContract]
    public class MarsChallengerState
    {
        /// <summary>
        /// Current rover state
        /// </summary>
        [DataMember]
        public RoverState State { get; set; }

        /// <summary>
        /// Last received comlink data
        /// </summary>
        [DataMember]
        public comlink.MarsComLinkState ComLinkData { get; set; }

        /// <summary>
        /// Last received spectrometer data
        /// </summary>
        [DataMember]
        public spectro.MarsSpectrometerState SpectrometerData { get;
set; }

        /// <summary>
        /// Last image processing result for front cam
        /// </summary>
        [DataMember]
        public processor.ImageResult FrontCamData { get; set; }

        /// <summary>
        /// Last image processing result for rear cam
        /// </summary>
        [DataMember]
        public processor.ImageResult RearCamData { get; set; }
```

```
        /// <summary>
        /// Last image processing result for nav cam
        /// </summary>
        [DataMember]
        public processor.ImageResult NavCamData { get; set; }

        /// <summary>
        /// Last image processing result for pan cam
        /// </summary>
        [DataMember]
        public processor.ImageResult PanCamData { get; set; }
    }

    /// <summary>
    /// Mars Rover States
    /// </summary>
    [DataContract]
    public enum RoverState
    {
        /// <summary>
        /// Intitialize the robot
        /// </summary>
        Initialize,

        /// <summary>
        /// Goes to certain position
        /// </summary>
        Navigate,


        /// <summary>
        /// Looks for a rock
        /// </summary>
        RockFinder,


        /// <summary>
        /// Robot Stuck
        /// </summary>
        Stuck,



        /// <summary>
        /// HeatShield search
        /// </summary>
        HeatShield,

        /// <summary>
        /// HeatShield approximation
        /// </summary>
        HeatFound,


        /// <summary>
        /// Mission is over
        /// </summary>
        EndMission,
```

```
    }

    #region Service Operations

    [ServicePort]
    public class MarsChallengerOperations : PortSet<DsspDefaultLookup,
DsspDefaultDrop, Get, Initialize, HttpGet>
    {
    }

    public class Get : Get<GetRequestType,
PortSet<MarsChallengerState, Fault>>
    {
        public Get() : base() { }
        public Get(GetRequestType body) : base(body) { }
        public Get(GetRequestType body, PortSet<MarsChallengerState,
Fault> responsePort) : base(body, responsePort) { }
    }

    public class Initialize : Update<InitializeRequest,
PortSet<DefaultUpdateResponseType, Fault>>
    {
        public Initialize() : base() { }
        public Initialize(InitializeRequest body) : base(body) { }
        public Initialize(InitializeRequest body,
PortSet<DefaultUpdateResponseType, Fault> responsePort) : base(body,
responsePort) { }
    }

    [DataContract]
    public class InitializeRequest
    {
    }

    #endregion
}
```

Fichero MarsChallenger.cs

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

using drive = Microsoft.Robotics.Services.Drive.Proxy;
using processor =
Microsoft.Robotics.RoboChamps.MarsChallenger.ImageProcessing;
using comlink = Microsoft.Robotics.RoboChamps.MarsComLink.Proxy;
using spectro = Microsoft.Robotics.RoboChamps.MarsSpectrometer.Proxy;
using pantilt = SimplySim.Robotics.PanTilt.Proxy;
using arm = Microsoft.Robotics.Services.ArticulatedArm.Proxy;



namespace Microsoft.Robotics.RoboChamps.MarsChallenger
{

    [DisplayName("MarsChallenger")]
    [Description("The Mars Challenger sample Service")]
    [Contract(Contract.Identifier)]
    public class MarsChallengerService : DsspServiceBase
    {
        /// <summary>
        /// Constant vector for articulated arm operation
        /// </summary>
        private static Vector3 Twist = new Vector3(1, 0, 0);

        /// <summary>
        /// Constants for arm joint names
        /// </summary>
        private const string ShoulderHJointName = "ShoulderH";
        private const string ShoulderVJointName = "ShoulderV";
        private const string ElbowJointName = "Elbow";
        private const string WristHJointName = "WristH";
        private const string WristVJointName = "WristV";


        /// <summary>
        /// Next position we are moving to
        /// </summary>
        private comlink.Position target;

        /// <summary>
        /// Potency to give to the wheels
        /// </summary>
        private double leftWheel=0;
        private double rightWheel=0;
```

```
        /// <summary>
        /// True: robot arrived to the sample zone where it was moving
to
        /// </summary>
        private bool found = false;

        /// <summary>
        /// Sample Zone the robot is moving to:
        /// #0: sample zone 1
        /// #1: sample zone 2
        /// ...
        /// #5: sample zone 6
        /// </summary>
        private int targetNum = 0;


        bool zoneNotified;

        /// <summary>
        /// True: the robot has the arm ready to analyze.
        /// </summary>
        bool finding = true;

        /// <summary>
        /// Counts the consecutive times the robot is in Navigate
        /// State but remains in the same place
        /// </summary>
        private int stopCounter = 0;

        /// <summary>
        ///
        /// </summary>
        MarsComLink.Proxy.Area prevInfo = new
Microsoft.Robotics.RoboChamps.MarsComLink.Proxy.Area();

        /// <summary>
        /// Orientation of the robot in the previous loop iteration
        /// </summary>
        float prevYaw = 0;

        /// <summary>
        ///
        /// </summary>
        private int stuckCounter = 0;

        /// <summary>
        /// Previous state of the robot
        /// </summary>
        RoverState prevState = RoverState.Initialize;

        /// <summary>
        /// True: last rocks seen rightwards
        /// </summary>
        bool spinRight = false;

        /// <summary>
        /// Sequence of positions the robot shall follow in order to
arrive
        /// to certain destination
        /// </summary>
```

```
        private ArrayList path = new ArrayList();

        /// <summary>
        /// Step of current path the robot is going to
        /// </summary>
        int pathNum;

        /// <summary>
        /// Last position the robot got stuck in
        /// </summary>
        comlink.Position lastStuck = new
Microsoft.Robotics.RoboChamps.MarsComLink.Proxy.Position();

        /// <summary>
        /// Number of times the robot has gotten stuck in the same
position
        /// </summary>
        int stuckTimes = 0;

        /// <summary>
        /// Number of consecutive iterations the robot is spinning
looking for rocks.
        /// </summary>
        int spinCounter = 0;


        /// <summary>
        /// Iterations passed since current path was last revised
        /// </summary>
        private int pathRev = 0;


        /// <summary>
        /// Robot is inside the crater
        /// </summary>
        private bool inCrater = false;

        /// <summary>
        /// Wheels potency in previous iteration
        /// </summary>
        double prevLeft = 0;
        double prevRight = 0;


        /// <summary>
        /// True: sample 5 analyzed, heading to sample zone 6 via
zones 4 and 1.
        /// </summary>
        bool finalPath = false;

        /// <summary>
        /// True: current navigation heads towards shield search
        /// </summary>
        private bool shieldSearch;

        /// <summary>
        /// Positions near which the heat shield can be located
        /// </summary>
        comlink.Area[] search;

        /// <summary>
```

```
        /// Number of consecutive times the service is in *******
        /// </summary>
        private int shieldCounter;

        /// <summary>
        /// True: the heat shield has been succesfully located and its
area visited
        /// </summary>
        private bool shieldLocated;

        /// <summary>
        /// True: the six different samples have been analyzed
        /// </summary>
        private bool rocksAnalyzed;



        /// <summary>
        /// Service state.
        /// You can view current state with a browser
http://localhost:50000/marschallenger
        /// </summary>
        [ServiceState]
        private MarsChallengerState _state = new
MarsChallengerState();

        /// <summary>
        /// Service operation port
        /// </summary>
        [ServicePort("/MarsChallenger", AllowMultipleInstances =
false)]
        private MarsChallengerOperations _mainPort = new
MarsChallengerOperations();

        #region Partners

        [Partner("ComLink", Contract = comlink.Contract.Identifier,
CreationPolicy = PartnerCreationPolicy.UsePartnerListEntry)]
        private comlink.MarsComLinkOperations _comlinkPort = new
comlink.MarsComLinkOperations();
        private comlink.MarsComLinkOperations _comlinkNotify = new
comlink.MarsComLinkOperations();

        [Partner("Spectro", Contract = spectro.Contract.Identifier,
CreationPolicy = PartnerCreationPolicy.UsePartnerListEntry)]
        private spectro.MarsSpectrometerOperations _spectroPort = new
spectro.MarsSpectrometerOperations();
        private spectro.MarsSpectrometerOperations _spectroNotify =
new spectro.MarsSpectrometerOperations();

        [Partner("PanCamProcessor", Contract =
processor.Contract.Identifier, CreationPolicy =
PartnerCreationPolicy.UsePartnerListEntry)]
        private processor.ImageProcessingOperations _pancamPort = new
processor.ImageProcessingOperations();
        private processor.ImageProcessingOperations _pancamNotify =
new processor.ImageProcessingOperations();

        [Partner("NavCamProcessor", Contract =
processor.Contract.Identifier, CreationPolicy =
PartnerCreationPolicy.UsePartnerListEntry)]
```

```
        private processor.ImageProcessingOperations _navcamPort = new
processor.ImageProcessingOperations();
        private processor.ImageProcessingOperations _navcamNotify =
new processor.ImageProcessingOperations();

        [Partner("FrontCamProcessor", Contract =
processor.Contract.Identifier, CreationPolicy =
PartnerCreationPolicy.UsePartnerListEntry)]
        private processor.ImageProcessingOperations _frontcamPort =
new processor.ImageProcessingOperations();
        private processor.ImageProcessingOperations _frontcamNotify =
new processor.ImageProcessingOperations();

        [Partner("RearCamProcessor", Contract =
processor.Contract.Identifier, CreationPolicy =
PartnerCreationPolicy.UsePartnerListEntry)]
        private processor.ImageProcessingOperations _rearcamPort = new
processor.ImageProcessingOperations();
        private processor.ImageProcessingOperations _rearcamNotify =
new processor.ImageProcessingOperations();

        [Partner("Drive", Contract = drive.Contract.Identifier,
CreationPolicy = PartnerCreationPolicy.UsePartnerListEntry)]
        private drive.DriveOperations _drivePort = new
drive.DriveOperations();

        [Partner("Head", Contract = pantilt.Contract.Identifier,
CreationPolicy = PartnerCreationPolicy.UsePartnerListEntry)]
        private pantilt.PanTiltOperations _headPort = new
pantilt.PanTiltOperations();

        [Partner("Arm", Contract = arm.Contract.Identifier,
CreationPolicy = PartnerCreationPolicy.UsePartnerListEntry)]
        private arm.ArticulatedArmOperations _armPort = new
arm.ArticulatedArmOperations();


        #endregion

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

        protected override void Start()
        {

            base.Start();
            _mainPort.Post(new Initialize());
        }

        #region Initialization

        [ServiceHandler(ServiceHandlerBehavior.Exclusive)]
        public IEnumerator<ITask> InitializeHandler(Initialize
initialize)
        {
            // Get sensors initial state
            yield return Arbiter.Choice(_comlinkPort.Get(), (success)
=> _state.ComLinkData = success, LogError);
```

```
            yield return Arbiter.Choice(_spectroPort.Get(), (success)
=> _state.SpectrometerData = success, LogError);

            // Adding handlers for sensors and camera processor
notifications
            MainPortInterleave.CombineWith(
                Arbiter.Interleave(
                    new TeardownReceiverGroup(),
                    new ExclusiveReceiverGroup
                    (
                        Arbiter.Receive<comlink.Replace>(true,
_comlinkNotify, ComlinkReplaceHandler),
                        Arbiter.Receive<spectro.Replace>(true,
_spectroNotify, SpectroReplaceHandler),
                        Arbiter.Receive<processor.ProcessImage>(true,
_pancamNotify, PanCamProcessImageHandler),
                        Arbiter.Receive<processor.ProcessImage>(true,
_navcamNotify, NavCamProcessImageHandler),
                        Arbiter.Receive<processor.ProcessImage>(true,
_frontcamNotify, FrontCamProcessImageHandler),
                        Arbiter.Receive<processor.ProcessImage>(true,
_rearcamNotify, RearCamProcessImageHandler)
                    ),
                    new ConcurrentReceiverGroup()));

            // Subscribing
            yield return Arbiter.Choice(
                _comlinkPort.Subscribe(_comlinkNotify,
typeof(comlink.Replace)),
                (success) => { },
                (fault) => LogError(fault));

            yield return Arbiter.Choice(
                _spectroPort.Subscribe(_spectroNotify,
typeof(spectro.Replace)),
                (success) => { },
                (fault) => LogError(fault));

            yield return Arbiter.Choice(
                _pancamPort.Subscribe(_pancamNotify,
typeof(processor.ProcessImage)),
                (success) => { },
                (fault) => LogError(fault));

            yield return Arbiter.Choice(
                _navcamPort.Subscribe(_navcamNotify,
typeof(processor.ProcessImage)),
                (success) => { },
                (fault) => LogError(fault));

            yield return Arbiter.Choice(
                _frontcamPort.Subscribe(_frontcamNotify,
typeof(processor.ProcessImage)),
                (success) => { },
                (fault) => LogError(fault));

            yield return Arbiter.Choice(
                _rearcamPort.Subscribe(_rearcamNotify,
typeof(processor.ProcessImage)),
                (success) => { },
                (fault) => LogError(fault));
```

```
            _state.State = RoverState.Initialize;
            shieldCounter = 0;
            shieldSearch = false;
            rocksAnalyzed = false;
            shieldLocated = false;
            // Start the main loop
            RearmMainLoop(500);
        }

        #endregion

        #region Main Loop

        /// <summary>
        /// Used to keep a position in this sample
        /// </summary>
        private comlink.Position _lastPosition;

        private IEnumerator<ITask> MainLoopHandler(DateTime time)
        {
            // In the loop, when we send a command to the robot we use
            // Arbiter.Choice for waiting the command response.
            // You can just call the command without the Choice is you
            // don't care about the answer

            if (!inCrater && _state.ComLinkData.Position.Y < 0)
                inCrater = true;


            // You can retreive current waypoint if you are in a
waypoint area
            if(_state.State!= RoverState.Initialize )
            if ( _state.ComLinkData.CurrentArea != null)
            {


if(_state.ComLinkData.CurrentArea.Name.Contains("shield")){
                       // LogInfo(LogGroups.Console, String.Format("Shield
Located"));
                        shieldLocated=true;
                }

                String s = (targetNum + 1).ToString();

                    if
(_state.ComLinkData.CurrentArea.Name.Contains(s))
                    {
                        if (inCrater && targetNum == 0)
                        {
                            found = true;
                        }

                        if (finalPath && targetNum == 3)
                        {
                            found = true;
                        }

                        if (finalPath && targetNum == 0)
                        {
                             found = true;
```

```
                        }


                        if (!zoneNotified)
                        {
                            setFindingPosition();
                            yield return
Arbiter.Choice(_headPort.SetTilt(-20), (success) => { }, LogError);
                            zoneNotified = true;
                        }

                    }
            }
            else
            {
                zoneNotified = false;


                if (_state.State == RoverState.RockFinder)
                {
                   // LogInfo(LogGroups.Console, String.Format("Robot
in no sample zone"));
                    target = (comlink.Position) path[path.Count - 1];
                    pathNum = path.Count - 1;
                    changeState(RoverState.Navigate);

                }

            }

            if (found)
            {
                //Sample zone the robot was heading towards reached


                //Robot in navigation physical position

                yield return Arbiter.Choice(

    _armPort.SetJointTargetPose(ShoulderHJointName, new Vector3(), new
AxisAngle(Twist, ToRadians(0))),
    (success) => { }, LogError);
                yield return Arbiter.Choice(

                    _armPort.SetJointTargetPose(ShoulderVJointName,
new Vector3(), new AxisAngle(Twist, ToRadians(15))),
                    (success) => { }, LogError);
                yield return Arbiter.Choice(

                    _armPort.SetJointTargetPose(ElbowJointName, new
Vector3(), new AxisAngle(Twist, ToRadians(0))),
                    (success) => { }, LogError);
                yield return Arbiter.Choice(

                    _armPort.SetJointTargetPose(WristHJointName, new
Vector3(), new AxisAngle(Twist, ToRadians(-90))),
                    (success) => { }, LogError);
                yield return Arbiter.Choice(
                    _armPort.SetJointTargetPose(WristVJointName, new
Vector3(), new AxisAngle(Twist, ToRadians(0))),
                    (success) => { }, LogError);
```

```
                    targetNum++;

                    if (!shieldSearch)
                    {//If there are samples not yet analyzed

                        if (targetNum == 1 && inCrater)
                        {
                            //Finally got out of the crater
                            targetNum = 3;
                           inCrater = false;
                        }

                        if (targetNum == 3 && inCrater)
                        {
                            //From sample zone 3 to 4 we pass for zone
1 first

                            // due to the high slope in the path
between the

                            //first two.
                            targetNum = 0;

                        }

                           if (targetNum == 5)
                           {
                               //From sample zone 5 to 6 we pass for
zones 4 and 1 first

                               // in order to avoid the crater.
                               targetNum = 3;
                               finalPath = true;
                           }

                        if (targetNum == 4 && finalPath)
                        {

                            targetNum = 0;
                        }

                        if (targetNum == 1 && finalPath)
                        {
                            targetNum = 5;
                            finalPath = false;
                        }

                        found = false;

                        yield return
Arbiter.Choice(_headPort.SetPan(0), (success) => { }, LogError);
                        yield return
Arbiter.Choice(_headPort.SetTilt(0), (success) => { }, LogError);
                        if (targetNum <
_state.ComLinkData.Areas.Length)
                        {
                            //If there are still samples not analyzed
                            path =
getPath(_state.ComLinkData.Position,
_state.ComLinkData.Areas[targetNum].Position);
                            pathNum = 0;
                            target = (comlink.Position)path[0];
```

```
                                setNormalPosition();
                                changeState(RoverState.Navigate);
                            }
                            else
                            {
                                //Six samples analyzed
                                rocksAnalyzed = true;
                                shieldSearch = true;

                                //ShieldArea to patrol in search of the
heat shield
                                search = new
Microsoft.Robotics.RoboChamps.MarsComLink.Proxy.Area[9];
                                search[0] = new
Microsoft.Robotics.RoboChamps.MarsComLink.Proxy.Area();
                                search[0].Position = new
Microsoft.Robotics.RoboChamps.MarsComLink.Proxy.Position();
                                search[0].Position.X = 140;
                                search[0].Position.Z = 10;
                                search[1] = new
Microsoft.Robotics.RoboChamps.MarsComLink.Proxy.Area();
                                search[1].Position = new
Microsoft.Robotics.RoboChamps.MarsComLink.Proxy.Position();
                                search[1].Position.X = 140;
                                search[1].Position.Z = 160;
                                search[2] = new
Microsoft.Robotics.RoboChamps.MarsComLink.Proxy.Area();
                                search[2].Position = new
Microsoft.Robotics.RoboChamps.MarsComLink.Proxy.Position();
                                search[2].Position.X = 0;
                                search[2].Position.Z = 133;
                                search[3] = new
Microsoft.Robotics.RoboChamps.MarsComLink.Proxy.Area();
                                search[3].Position = new
Microsoft.Robotics.RoboChamps.MarsComLink.Proxy.Position();
                                search[3].Position.X = -140;
                                search[3].Position.Z = 133;
                                search[4] = new
Microsoft.Robotics.RoboChamps.MarsComLink.Proxy.Area();
                                search[4].Position = new
Microsoft.Robotics.RoboChamps.MarsComLink.Proxy.Position();
                                search[4].Position.X = -140;
                                search[4].Position.Z = 0;
                                search[5] = new
Microsoft.Robotics.RoboChamps.MarsComLink.Proxy.Area();
                                search[5].Position = new
Microsoft.Robotics.RoboChamps.MarsComLink.Proxy.Position();
                                search[5].Position.X = -140;
                                search[5].Position.Z = -160;
                                search[6] = new
Microsoft.Robotics.RoboChamps.MarsComLink.Proxy.Area();
                                search[6].Position = new
Microsoft.Robotics.RoboChamps.MarsComLink.Proxy.Position();
                                search[6].Position.X = -110;
                                search[6].Position.Z = -160;
                                search[7] = new
Microsoft.Robotics.RoboChamps.MarsComLink.Proxy.Area();
                                search[7].Position = new
Microsoft.Robotics.RoboChamps.MarsComLink.Proxy.Position();
                                search[7].Position.X = -110;
```

```
                            search[7].Position.Z = 110;
                            search[8] = new
Microsoft.Robotics.RoboChamps.MarsComLink.Proxy.Area();
                            search[8].Position = new
Microsoft.Robotics.RoboChamps.MarsComLink.Proxy.Position();
                            search[8].Position.X = 110;
                            search[8].Position.Z = 110;
                            changeState(RoverState.Navigate);
                            targetNum = 0;
                        }

                    }
                    else
                    {
                        //Looking for heat shield, visit next patrol
area

                        found = false;
                        yield return
Arbiter.Choice(_headPort.SetPan(0), (success) => { }, LogError);
                        yield return
Arbiter.Choice(_headPort.SetTilt(0), (success) => { }, LogError);


                        if (targetNum >= search.Length)
                        {
                            //If all patrol area visited, start again.
                            targetNum = 0;
                        }
                            path =
getPath(_state.ComLinkData.Position, search[targetNum].Position);
                            pathNum = 0;
                            target = (comlink.Position)path[0];

                            setNormalPosition();
                            changeState(RoverState.Navigate);



                    }








            }

            if (shieldLocated && rocksAnalyzed)
            {
                //Objectives reached, CONGRATULATIONS!!
                changeState(RoverState.EndMission);
            }


            switch (_state.State)
            {
                case RoverState.Initialize:
                    lastStuck.X = 0;
                    lastStuck.Y = 0;
```

```
                    if(finding)
                     setNormalPosition();

                    yield return Arbiter.Choice(_headPort.SetPan(0),
(success) => { }, LogError);
                    yield return Arbiter.Choice(_headPort.SetTilt(0),
(success) => { }, LogError);
                    //Elaborate path to sample zone 1
                    path = getPath(_state.ComLinkData.Position,
_state.ComLinkData.Areas[targetNum].Position);
                    pathNum = 0;
                    rightWheel = 0;
                    leftWheel = 0;
                    rocksAnalyzed = false;
                    shieldLocated = false;
                    shieldCounter = 0;
                    shieldSearch = false;
                    target = (comlink.Position) path[0];


                    changeState(RoverState.Navigate);
                    _lastPosition = _state.ComLinkData.Position;
                    // Rearm main loop after 1s
                    RearmMainLoop(1000);
                    break;


                case RoverState.HeatShield:

                    shieldCounter++;

                    if (shieldCounter == 1)
                    {
                        yield return
Arbiter.Choice(_headPort.SetPan(20), (success) => { }, LogError);
                    }
                    else if (shieldCounter == 2)
                    {
                        yield return
Arbiter.Choice(_headPort.SetPan(40), (success) => { }, LogError);
                    }
                    else if (shieldCounter == 3)
                    {
                        yield return
Arbiter.Choice(_headPort.SetPan(60), (success) => { }, LogError);
                    }
                    else if (shieldCounter == 4)
                    {
                        yield return
Arbiter.Choice(_headPort.SetPan(75), (success) => { }, LogError);
                    }
                    else if (shieldCounter == 5)
                    {
                        yield return
Arbiter.Choice(_headPort.SetPan(90), (success) => { }, LogError);
                    }
                    else if (shieldCounter == 6)
                    {
                        yield return
Arbiter.Choice(_headPort.SetPan(0), (success) => { }, LogError);
```

```
                    }
                    else if (shieldCounter == 7)
                    {
                        yield return Arbiter.Choice(_headPort.SetPan(-
20), (success) => { }, LogError);
                    }
                    else if (shieldCounter == 8)
                    {
                        yield return Arbiter.Choice(_headPort.SetPan(-
40), (success) => { }, LogError);
                    }
                    else if (shieldCounter == 9)
                    {
                        yield return Arbiter.Choice(_headPort.SetPan(-
60), (success) => { }, LogError);
                    }
                    else if (shieldCounter == 10)
                    {
                        yield return Arbiter.Choice(_headPort.SetPan(-
75), (success) => { }, LogError);
                    }
                    else if (shieldCounter == 11)
                    {
                        yield return Arbiter.Choice(_headPort.SetPan(-
90), (success) => { }, LogError);
                    }
                    else if (shieldCounter == 12)
                    {
                        yield return
Arbiter.Choice(_headPort.SetPan(0), (success) => { }, LogError);
                        pathNum++;
                        if (pathNum >= (path.Count - 1))
                        {
                            // Reached a patrol location, head towards
next.
                                  found = true;
                        }
                        else
                        {

                            //More steps left in the path, move to
next one.
                            target = (comlink.Position)path[pathNum];
                        }

                        shieldCounter = 0;
                        changeState(RoverState.Navigate);
                    }

                    if (_state.PanCamData.ShieldArea > 10)
                    {
                        //If shield visualized

                        yield return
Arbiter.Choice(_headPort.SetPan(0), (success) => { }, LogError);

                        //Take action depending on the relative
position the shield was sighted.
                        if (shieldCounter <= 2)
                        {
```

```
                            yield return
Arbiter.Choice(_drivePort.SetDrivePower(0.1,-0.1), (success) => { },
LogError);

                        }
                        else if (shieldCounter <= 5)
                        {
                            yield return
Arbiter.Choice(_drivePort.SetDrivePower(0.2, -0.2), (success) => { },
LogError);
                        }
                        else if (shieldCounter == 6 ||
shieldCounter==12)
                        {
                            yield return
Arbiter.Choice(_drivePort.SetDrivePower(1, 1), (success) => { },
LogError);
                        }
                        else if (shieldCounter <= 8)
                        {
                            yield return
Arbiter.Choice(_drivePort.SetDrivePower(-0.1, 0.1), (success) => { },
LogError);
                        }
                        else if (shieldCounter <= 11)
                        {
                            yield return
Arbiter.Choice(_drivePort.SetDrivePower(-0.2, 0.2), (success) => { },
LogError);
                        }

                        changeState(RoverState.HeatFound);
                        shieldCounter = 0;
                    }

                    RearmMainLoop(500);
                    break;



            case RoverState.HeatFound:

                    //Heat shield located

                    yield return Arbiter.Choice(_headPort.SetPan(0),
(success) => { }, LogError);
                    shieldCounter++;
                    if (_state.PanCamData.ShieldArea > 10)
                    {
                        //If shield in sight of panoramic camera, move
straight ahead.
                        yield return
Arbiter.Choice(_drivePort.SetDrivePower(1, 1), (success) => { },
LogError);
                    }
                    else if (_state.NavCamData.ShieldArea > 10)
                    {
                        //If shield in sight of navigation camera,
move ahead or rotate

                        if (_state.NavCamData.XMean <= 42)
```

```
                        {
                            yield return
Arbiter.Choice(_drivePort.SetDrivePower(-0.1, 0.1), (success) => { },
LogError);
                        }
                        else if (_state.NavCamData.XMean <= 84)
                        {
                            yield return
Arbiter.Choice(_drivePort.SetDrivePower(1, 1), (success) => { },
LogError);
                        }
                        else
                        {
                            yield return
Arbiter.Choice(_drivePort.SetDrivePower(0.1, -0.1), (success) => { },
LogError);
                        }

                    }
                    else
                    {
                        //Shield out of sight

                        if (shieldCounter > 10)
                        {
                            //Ten iteration without sight of the
shield, return to patrol movement
                            shieldCounter = 0;
                            changeState(RoverState.HeatShield);
                        }
                    }



                    RearmMainLoop(500);
                    break;

                case RoverState.Navigate:

                    pathRev++;
                    if (pathRev > 20)
                    {
                        //Revise current path every 20 iterations
                        pathRevision();

                    }

                    if ( Distance(_lastPosition,
_state.ComLinkData.Position) < 0.2)
                    { //If little location movement

                        if (noRotation())
                        {//and not even rotation, counter increases
                            stopCounter++;
                        }
                        else
                        {
                            //Orientation changed
                            prevYaw = _state.ComLinkData.Yaw;
                            stopCounter = 0;
                        }
```

```
                        }
                        else
                        {
                            prevYaw = _state.ComLinkData.Yaw;
                            _lastPosition = _state.ComLinkData.Position;
                            stopCounter = 0;
                        }

                        if (stopCounter > 50)
                        {
                            //If no movement for a lot of iterations,
change to Stuck State

                            if (Distance(lastStuck,
_state.ComLinkData.Position) < 1)
                            {
                                //If we previously got stuck in that
position
                                stuckTimes++;
                            }
                            else
                            {
                                stuckTimes = 0;
                                lastStuck = _state.ComLinkData.Position;
                            }

                            changeState(RoverState.Stuck);
                        }
                        else
                        {


                            if (Math.Abs(_state.ComLinkData.Pitch) > 90)
                            {
                                //If robot is wheels-up head-down, mission
ended
                                changeState(RoverState.EndMission);
                            }

                            //Vertical distance between current position
and target position
                            double verticalD =
Math.Abs(_state.ComLinkData.Position.Z - target.Z);

                            //Horizontal distance between current position
and target position
                            double horizontalD =
Math.Abs(_state.ComLinkData.Position.X - target.X);


                            //Orientation is set
                            if (_state.ComLinkData.Position.Z < target.Z)
                            {
                                verticalD *= -1;
                            }


                            if (_state.ComLinkData.Position.X > target.X)
                            {
                                horizontalD *= -1;
```

```
                        }


                        //If far from destination, get back to normal
position
                        if (finding && (Math.Abs(verticalD) > 10 ||
Math.Abs(horizontalD) > 10))
                            setNormalPosition();


                        if ((pathNum < (path.Count - 1) &&
(Math.Abs(horizontalD) > 3 || Math.Abs(verticalD) > 3)) ||
                            (pathNum >= (path.Count - 1) &&
(Math.Abs(horizontalD) > 1 || Math.Abs(verticalD) > 1)))
                        {//If not yet in the location the robot goes
to

                            /*//If little vertical distance, it is
assumed zero.
                            if (Math.Abs(verticalD) < 5)
                            {
                                verticalD = 0;
                            }

                            //If little horizontal distance, it is
assumed zero.
                            if (Math.Abs(horizontalD) < 5)
                            {
                                horizontalD = 0;
                            }*/

                            rightWheel = 1;
                            leftWheel = 1;
                            if (horizontalD > 0 && verticalD > 0)
                            {
                                //Target in an upper-right location
                                spinRight = true;

                                //Robot rotates depending on its
actual Yaw (orientation) to head
                                //target location.
                                if (_state.ComLinkData.Yaw < 90 &&
_state.ComLinkData.Yaw > -45)
                                {

                                    rightWheel = 0.25;
                                    leftWheel = -0.25;
                                }
                                else if (_state.ComLinkData.Yaw < -45)
                                {
                                    rightWheel = -0.25;
                                    leftWheel = 0.25;
                                }
                                else
                                {
                                    //The robot orientation is close
to the desired one
```

```
                                    //Slope is calculated in order to
determine the desired movement
                                    double slope = Math.Abs(verticalD
/ horizontalD);
                                    if (slope >= 0.5 && slope <= 2)
                                    {
                                        if (_state.ComLinkData.Yaw >=
90 && _state.ComLinkData.Yaw < 113)
                                        {
                                            rightWheel = 1;
                                            leftWheel = 0.8;
                                        }
                                        else if
(_state.ComLinkData.Yaw >= 157 && _state.ComLinkData.Yaw <= 180)
                                        {
                                            rightWheel = 0.8;
                                            leftWheel = 1;
                                        }
                                    }
                                    else if (slope < 0.5)
                                    {
                                        if (_state.ComLinkData.Yaw >=
157 && _state.ComLinkData.Yaw <= 180)
                                        {
                                            rightWheel = -0.25;
                                            leftWheel = 0.25;
                                        }
                                        else if
(_state.ComLinkData.Yaw >= 113 && _state.ComLinkData.Yaw <= 157)
                                        {
                                            rightWheel = 0.8;
                                            leftWheel = 1;
                                        }
                                    }
                                    else if (slope > 2)
                                    {
                                        if (_state.ComLinkData.Yaw >=
90 && _state.ComLinkData.Yaw <= 113)
                                        {
                                            rightWheel = 0.25;
                                            leftWheel = -0.25;
                                        }
                                        else if
(_state.ComLinkData.Yaw >= 113 && _state.ComLinkData.Yaw <= 157)
                                        {
                                            rightWheel = 1;
                                            leftWheel = 0.8;
                                        }
                                    }

                                }


                            }
                            else if (horizontalD > 0 && verticalD < 0)
                            {

                                //Target in a lower-right location
                                spinRight = true;
```

```
                                if (Math.Abs(_state.ComLinkData.Yaw) >
135 || _state.ComLinkData.Yaw > 90)
                                {
                                    rightWheel = -0.25;
                                    leftWheel = 0.25;
                                }
                                else if (_state.ComLinkData.Yaw < 0)
                                {
                                    rightWheel = 0.25;
                                    leftWheel = -0.25;
                                }
                                else
                                {
                                    double slope = Math.Abs(verticalD
/ horizontalD);
                                    if (slope >= 0.5 && slope <= 2)
                                    {
                                        if (_state.ComLinkData.Yaw >=
0 && _state.ComLinkData.Yaw < 23)
                                        {
                                            rightWheel = 1;
                                            leftWheel = 0.8;
                                        }
                                        else if
(_state.ComLinkData.Yaw >= 67 && _state.ComLinkData.Yaw <= 90)
                                        {
                                            rightWheel = 0.8;
                                            leftWheel = 1;
                                        }
                                    }
                                    else if (slope < 0.5)
                                    {
                                        if (_state.ComLinkData.Yaw >=
67 && _state.ComLinkData.Yaw <= 90)
                                        {
                                            rightWheel = -0.25;
                                            leftWheel = 0.25;
                                        }
                                        else if
(_state.ComLinkData.Yaw >= 45 && _state.ComLinkData.Yaw <= 67)
                                        {
                                            rightWheel = 0.8;
                                            leftWheel = 1;
                                        }
                                    }
                                    else if (slope > 2)
                                    {
                                        if (_state.ComLinkData.Yaw >=
0 && _state.ComLinkData.Yaw <= 45)
                                        {
                                            rightWheel = 0.25;
                                            leftWheel = -0.25;
                                        }
                                        else if
(_state.ComLinkData.Yaw >= 45 && _state.ComLinkData.Yaw <= 67)
                                        {
                                            rightWheel = 1;
                                            leftWheel = 0.8;
                                        }
                                    }
```

```
                }

            }
            else if (horizontalD < 0 && verticalD > 0)
            {

                //Target in an upper-left location
                spinRight = false;
                   if (_state.ComLinkData.Yaw > 45)
                {
                    rightWheel = 0.25;
                    leftWheel = -0.25;
                }
                else if (_state.ComLinkData.Yaw > -90)
                {
                    rightWheel = -0.25;
                    leftWheel = 0.25;
                }
                else
                {
                    double slope = Math.Abs(verticalD
/ horizontalD);

                    if (slope >= 0.5 && slope <= 2)
                    {
                        if (_state.ComLinkData.Yaw >=
-180 && _state.ComLinkData.Yaw <= -157)
                        {
                            rightWheel = 1;
                            leftWheel = 0.8;
                        }
                        else if
(_state.ComLinkData.Yaw >= -135 && _state.ComLinkData.Yaw <= -90)
                        {
                            rightWheel = 0.8;
                            leftWheel = 1;
                        }
                    }
                    else if (slope < 0.5)
                    {
                        if (_state.ComLinkData.Yaw >=
-113 && _state.ComLinkData.Yaw <= -90)
                        {
                            rightWheel = -0.25;
                            leftWheel = 0.25;
                        }
                        else if
(_state.ComLinkData.Yaw >= -157 && _state.ComLinkData.Yaw <= -113)
                        {
                            rightWheel = 0.8;
                            leftWheel = 1;
                        }
                    }
                    else if (slope > 2)
                    {
                        if (_state.ComLinkData.Yaw >=
-180 && _state.ComLinkData.Yaw <= -157)
                        {
                            rightWheel = 0.25;
                            leftWheel = -0.25;
                        }
```

```
                                        else if
(_state.ComLinkData.Yaw >= -157 && _state.ComLinkData.Yaw <= -113)
                                        {
                                            rightWheel = 1;
                                            leftWheel = 0.8;
                                        }
                                    }

                                }

                            }
                            else if (horizontalD < 0 && verticalD < 0)
                            {
                                //Target in a lower-left location

                                spinRight = false;
                                if (_state.ComLinkData.Yaw > 0 &&
_state.ComLinkData.Yaw < 135)
                                {
                                    rightWheel = -0.25;
                                    leftWheel = 0.25;
                                }
                                else if
(Math.Abs(_state.ComLinkData.Yaw) > 90)
                                {
                                    rightWheel = 0.25;
                                    leftWheel = -0.25;
                                }
                                else
                                {
                                    double slope = Math.Abs(verticalD
/ horizontalD);

                                    if (slope >= 0.5 && slope <= 2)
                                    {
                                        if (_state.ComLinkData.Yaw >=
-90 && _state.ComLinkData.Yaw < -67)
                                        {
                                            rightWheel = 1;
                                            leftWheel = 0.8;
                                        }
                                        else if
(_state.ComLinkData.Yaw >= -23 && _state.ComLinkData.Yaw <= 0)
                                        {
                                            rightWheel = 0.8;
                                            leftWheel = 1;
                                        }
                                    }
                                    else if (slope < 0.5)
                                    {
                                        if (_state.ComLinkData.Yaw >=
-23 && _state.ComLinkData.Yaw <= 0)
                                        {
                                            rightWheel = -0.25;
                                            leftWheel = 0.25;
                                        }
                                        else if
(_state.ComLinkData.Yaw >= -67 && _state.ComLinkData.Yaw <= -23)
                                        {
                                            rightWheel = 0.8;
                                            leftWheel = 1;
                                        }
```

```
                                    }
                                    else if (slope > 2)
                                    {
                                        if (_state.ComLinkData.Yaw >=
-90 && _state.ComLinkData.Yaw <= -67)
                                        {
                                            rightWheel = 0.25;
                                            leftWheel = -0.25;
                                        }
                                        else if
(_state.ComLinkData.Yaw >= -67 && _state.ComLinkData.Yaw <= -23)
                                        {
                                            rightWheel = 1;
                                            leftWheel = 0.8;
                                        }
                                    }

                                }

                            }
                            else if (horizontalD == 0 && verticalD !=
0)
                            {

                                if (verticalD > 0)
                                {
                                    //Target directly upwards

                                    if (_state.ComLinkData.Pitch > -
35)
                                        lookUp();
                                    else
                                    {//Avoid falling backwards
                                        rightWheel = -0.1;
                                        leftWheel = -0.1;

                                        target.X += 10;
                                        target.Z -= 20;
                                    }

                                }
                                else
                                {
                                    //Target directly downwards
                                    lookDown();
                                }
                            }
                            else if (verticalD == 0 && horizontalD !=
0)
                            {

                                if (horizontalD > 0)
                                {
                                    //Target directly at right
                                    spinRight = true;
                                    lookRight();
                                }
                                else
                                {
                                    //Target directly at left
                                    spinRight = false;
```

```
                lookLeft();
            }
        }

        if (prevLeft != leftWheel || prevRight !=
rightWheel)
        {
            //Wheels power varies

            if (_state.ComLinkData.Pitch > 15)
            {
               //If going down a slope, do not
decrease speed dramatically
               // in order to prevent rolling down
the slope.

                if (rightWheel > leftWheel)
                {
                    rightWheel = 0.7;
                    leftWheel = 0.4;

                }
                else
                {
                    rightWheel = 0.4;
                    leftWheel = 0.7;
                }


            }

            prevRight = rightWheel;
            prevLeft = leftWheel;

          //  LogInfo(LogGroups.Console,
String.Format("Current Movement: {0}  {1}", leftWheel, rightWheel));
        }

        if (_state.ComLinkData.Pitch < -45)
        {
            //If pitch very low, turn over risk

            if (rightWheel == leftWheel)
            {
                rightWheel = -0.5;
                leftWheel = -0.5;
            }
            else if (rightWheel > leftWheel)
            {
                rightWheel = -0.5;
                leftWheel = -0.7;
            }
            else if (rightWheel < leftWheel)
            {
                rightWheel = -0.7;
                leftWheel = -0.5;
            }
        }
```

```
                            yield return
Arbiter.Choice(_drivePort.SetDrivePower(leftWheel, rightWheel),
(success) => { }, LogError);

                        }
                        else
                        {//Arrived to target location


                            if (!shieldSearch)
                            {
                                //If not looking for shield
                                pathNum++;
                                if (pathNum >= (path.Count - 1))
                                {//If reached last location of a path

                                    if (!finding)
                                        setFindingPosition();

changeState(RoverState.RockFinder);

                                }
                                else
                                {//If there are more steps to take in
the path
                                    target =
(comlink.Position)path[pathNum];

                                }
                            }
                            else
                            {//Looking for heat shield. Stop and take
a look around.
                                yield return
Arbiter.Choice(_drivePort.SetDrivePower(0, 0), (success) => { },
LogError);
                                changeState(RoverState.HeatShield);
                            }

                        }

                        if (_state.NavCamData.RockArea > 5 && finding)
                        {
                            //If finding rocks and rocks at sight

                            changeState(RoverState.RockFinder);
                        }

                    }

                    RearmMainLoop(100);

                    break;


                case RoverState.RockFinder:
```

```
                    if (Distance(_lastPosition,
_state.ComLinkData.Position) < 0.1)
                    {
                        //If no movement

                        if (noRotation())
                        {
                            stopCounter++;
                        }
                        else
                        {

                            prevYaw = _state.ComLinkData.Yaw;
                            stopCounter = 0;
                        }

                    }
                    else
                    {
                        _lastPosition = _state.ComLinkData.Position;
                        prevYaw = _state.ComLinkData.Yaw;
                        stopCounter = 0;
                    }

                    if (stopCounter > 50)
                    {
                        if (Distance(lastStuck,
_state.ComLinkData.Position) < 1)
                        {
                            stuckTimes++;
                        }
                        else
                        {
                            stuckTimes = 0;
                            lastStuck = _state.ComLinkData.Position;
                        }
                        changeState(RoverState.Stuck);
                     }

                    //Prepare for analyzing rocks
                    if(!finding)
                        setFindingPosition();

                    if (_state.NavCamData.RockArea > 5)
                    {
                        //Rocks detected

                        spinCounter = 0;

                        if (_state.NavCamData.RockPosition == 2)
                        {
                            //Rocks at left side

                            spinRight = false;
                            yield return
Arbiter.Choice(_drivePort.SetDrivePower(0.0, 0.1), (success) => { },
LogError);
                        }
                        else if (_state.NavCamData.RockPosition == 4)
                        {
                            //Rocks at right side
```

```
                            spinRight = true;
                            yield return
Arbiter.Choice(_drivePort.SetDrivePower(0.2, 0.0), (success) => { },
LogError);

                        }
                        else if (_state.NavCamData.RockPosition == 3)
                            {
                                //Rocks straight ahead
                                yield return
Arbiter.Choice(_drivePort.SetDrivePower(0.2, 0.2), (success) => { },
LogError);
                            }
                            else if (_state.NavCamData.RockPosition ==
1)
                            {
                        //Not defined
                            }
                            else if (_state.NavCamData.RockPosition ==
5)
                            {
                         // Not defined
                            }
                        yield return
Arbiter.Choice(_headPort.SetTilt(-40), (success) => { }, LogError);

                    }
                    else
                    {
                        //Move head up to look for rocks
                        yield return
Arbiter.Choice(_headPort.SetTilt(-20), (success) => { }, LogError);
                        double verticalDist =
Math.Abs(_state.ComLinkData.Position.Z - target.Z);
                        double horizontalDist =
Math.Abs(_state.ComLinkData.Position.X - target.X);

                        if (horizontalDist < 5 && verticalDist < 5)
                        {//If close to sample location area, spin
around
                            spinCounter++;
                            if (spinRight)
                            {

                                  yield return
Arbiter.Choice(_drivePort.SetDrivePower(0.6, 0.2), (success) => { },
LogError);
                            }
                            else
                            {
                                yield return
Arbiter.Choice(_drivePort.SetDrivePower(0.2, 0.6), (success) => { },
LogError);
                            }

                        }
                        else
                        { //Far from the sample location area, go back
to Navigate and get closer

                            changeState(RoverState.Navigate);
```

```
                            spinCounter = 0;
                            path =
getPath(_state.ComLinkData.Position,
_state.ComLinkData.Areas[targetNum].Position);
                            target = (comlink.Position) path[0];
                            pathNum = 0;

                        }
                    }

                    if ( spinCounter > 20)
                    {
                        //If spent many iterations spinning and no
trace of a rock, go back to Navigate
                        yield return
Arbiter.Choice(_headPort.SetTilt(-20), (success) => { }, LogError);
                        target = (comlink.Position) path[path.Count -
1];

                        pathNum = path.Count - 1;
                        changeState(RoverState.Navigate);
                        spinCounter = 0;
                        yield return
Arbiter.Choice(_drivePort.SetDrivePower(0.3, 0.3), (success) => { },
LogError);

                    }

                    RearmMainLoop(100);

                    break;

                case RoverState.Stuck:
                    if (finding)
                        setNormalPosition();

                    if (stuckTimes < 3)
                    {//Stuck in current position for less than three
times
                        if (stuckCounter < 10)
                        {//Make a backwards move in order to be away
from stuck position

                            if (stuckCounter < 5)
                            {
                                stuckCounter++;
                                yield return
Arbiter.Choice(_drivePort.SetDrivePower(-1, -0.5), (success) => { },
LogError);
                            }
                            else
                            {
                                stuckCounter++;
                                yield return
Arbiter.Choice(_drivePort.SetDrivePower(-0.5, -1), (success) => { },
LogError);
                            }
                        }
                        else
                        {//Go back to Navigate
                            stuckCounter = 0;
```

```
                            stopCounter = 0;
                            changeState(prevState);
                            LogInfo(LogGroups.Console,
String.Format("Freed"));
                        }
                    }
                    else
                    {
                        //Stuck in the same place for many times,
recalculate a new path in order
                        //to avoid stuck causes.
                        stuckTimes = 0;
                        path =
getPathAvoidStuck(_state.ComLinkData.Position,
_state.ComLinkData.Areas[targetNum].Position);
                        pathNum = 0;
                        target = (comlink.Position) path[0];

                        changeState(prevState);
                    }
                    RearmMainLoop(200);

                    break;


                case RoverState.EndMission:
                    // Stop the rover
                    yield return
Arbiter.Choice(_drivePort.SetDrivePower(0, 0), (success) => { },
LogError);

                    // Send end mission message to Earth !
                    yield return
Arbiter.Choice(_comlinkPort.EndMission(), (success) => { }, LogError);

                    // Not rearm the loop
                    break;
            }
        }

        /// <summary>
        /// Rearms the main loop
        /// </summary>
        private void RearmMainLoop(int rearmAfter)
        {
            if (MainPortInterleave.ArbiterState !=
ArbiterTaskState.Done)
            {
                MainPortInterleave.CombineWith(Arbiter.Interleave(
                    new TeardownReceiverGroup(),
                    new ExclusiveReceiverGroup
                    (
                        Arbiter.ReceiveWithIterator<DateTime>(false,
TimeoutPort(rearmAfter), MainLoopHandler)
                    ),
                    new ConcurrentReceiverGroup()));
            }
        }

        #endregion
```

```
        #region Sensors Notifications Handlers

        private void ComlinkReplaceHandler(comlink.Replace replace)
        {
            // Keep only recent message
            if
(replace.Body.Timestamp.CompareTo(_state.ComLinkData.Timestamp) > 0)
            {
                _state.ComLinkData = replace.Body;
            }
        }

        private void SpectroReplaceHandler(spectro.Replace replace)
        {
            _state.SpectrometerData = replace.Body;

            if (replace.Body.SampleId == (targetNum+1))
            {//Analyzed a rock from the desired sample zone
                found = true;
            }
        }

        private void PanCamProcessImageHandler(processor.ProcessImage
process)
        {
            _state.PanCamData = process.Body;
        }

        private void NavCamProcessImageHandler(processor.ProcessImage
process)
        {
           _state.NavCamData = process.Body;
        }

        private void
FrontCamProcessImageHandler(processor.ProcessImage process)
        {
            _state.FrontCamData = process.Body;

        }

        private void RearCamProcessImageHandler(processor.ProcessImage
process)
        {
            _state.RearCamData = process.Body;
        }

        #endregion

        #region Helpers

        /// <summary>
        /// Converts degrees to radians
        /// </summary>
        private float ToRadians(float degree)
        {
            return (float)(degree * Math.PI / 180.0d);
        }

        /// <summary>
        /// Puts the arm in a position that allows rocks analysis
```

```
        /// </summary>
        private void setFindingPosition()
        {

            finding = true;
                _armPort.SetJointTargetPose(ShoulderHJointName, new
Vector3(), new AxisAngle(Twist, ToRadians(0)));

                _armPort.SetJointTargetPose(ShoulderVJointName, new
Vector3(), new AxisAngle(Twist, ToRadians(-15)));

                _armPort.SetJointTargetPose(ElbowJointName, new
Vector3(), new AxisAngle(Twist, ToRadians(0)));

                _armPort.SetJointTargetPose(WristHJointName, new
Vector3(), new AxisAngle(Twist, ToRadians(-90)));

                _armPort.SetJointTargetPose(WristVJointName, new
Vector3(), new AxisAngle(Twist, ToRadians(0)));


        }

        /// <summary>
        /// Puts the arm in a position that does not interfere in
robot movement
        /// </summary>
        private void setNormalPosition()
        {
            // Move the arm in a basic position
            finding = false;
            _armPort.SetJointTargetPose(ShoulderHJointName, new
Vector3(), new AxisAngle(Twist, ToRadians(-15)));

            _armPort.SetJointTargetPose(ShoulderVJointName, new
Vector3(), new AxisAngle(Twist, ToRadians(35)));

            _armPort.SetJointTargetPose(ElbowJointName, new Vector3(),
new AxisAngle(Twist, ToRadians(-10)));

            _armPort.SetJointTargetPose(WristHJointName, new
Vector3(), new AxisAngle(Twist, ToRadians(90)));

            _armPort.SetJointTargetPose(WristVJointName, new
Vector3(), new AxisAngle(Twist, ToRadians(90)));


        }

        /// <summary>
        /// Calculates the distance between two positions
        /// </summary>
        private float Distance(comlink.Position from, comlink.Position
to)
        {
            return (float)Math.Sqrt(Math.Pow(to.X - from.X, 2) +
Math.Pow(to.Y - from.Y, 2) + Math.Pow(to.Z - from.Z, 2));
        }

        /// <summary>
```

```
        /// Calculates the distance between two positions in X and Z
coordinates
        /// </summary>
        private float myDistance(comlink.Position from,
comlink.Position to)
        {
            return (float)Math.Sqrt(Math.Pow(to.X - from.X, 2) +
Math.Pow(to.Z - from.Z, 2));
        }

        /// <summary>
        /// Puts Robot looking at right ( Yaw in [85,95]
        /// </summary>
        private bool lookRight()
        {
            //Rotation of the robot depends on its current orientation
            if ((_state.ComLinkData.Yaw < 85)&&(
_state.ComLinkData.Yaw > -90))
            {
                    if ((_state.ComLinkData.Yaw < 85) &&
(_state.ComLinkData.Yaw >= 80))
                    {

                        this.rightWheel = 0.1;
                        this.leftWheel = -0.1;
                        return false;
                    }
                    else
                    {
                        this.rightWheel = 0.25;
                        this.leftWheel = -0.25;
                        return false;
                    }
            }
            else if ((_state.ComLinkData.Yaw > 95) &&
(_state.ComLinkData.Yaw < -90))
            {

                if ((_state.ComLinkData.Yaw > 95) &&
(_state.ComLinkData.Yaw <= 100))
                {
                    this.rightWheel = -0.1;
                    this.leftWheel = 0.1;
                    return false;
                }
                else
                {
                    this.rightWheel = -0.25;
                    this.leftWheel = 0.25;
                    return false;
                }
            }
            else
            {
                return true;
            }

        }

        /// <summary>
        /// Puts Robot looking at left ( Yaw in [-85,-95]
```

```
        /// </summary>
        private bool lookLeft()
        {
            //Rotation of the robot depends on its current orientation
            if ((_state.ComLinkData.Yaw < -95) ||
(_state.ComLinkData.Yaw > 90))
            {

                if ((_state.ComLinkData.Yaw < -95) ||
(_state.ComLinkData.Yaw >= -100))
                {
                    this.rightWheel = 0.1;
                    this.leftWheel = -0.1;
                    return false;
                }
                else
                {
                    this.rightWheel = 0.25;
                    this.leftWheel = -0.25;
                    return false;
                }
            }else  if((_state.ComLinkData.Yaw > -85)
&&(_state.ComLinkData.Yaw < 90))
            {

                if ((_state.ComLinkData.Yaw > -85) &&
(_state.ComLinkData.Yaw <= -80))
                {
                    this.rightWheel = -0.1;
                    this.leftWheel = 0.1;
                    return false;
                }
                else
                {

                    this.rightWheel = -0.25;
                    this.leftWheel = 0.25;
                    return false;
                }
            }
            else
            {
                return true;
            }

        }


        /// <summary>
        /// Puts Robot looking up ( Yaw in [175,180] or in [-175,-
180])
        /// </summary>
        private bool lookUp()
        {
            //Rotation of the robot depends on its current orientation
            if ((_state.ComLinkData.Yaw <
175)&&(_state.ComLinkData.Yaw>=0))
            {
                if ((_state.ComLinkData.Yaw >= 170))
                {
                    this.rightWheel = 0.1;
```

```
                    this.leftWheel = -0.1;
                    return false;
                }
                else
                {
                    this.rightWheel = 0.25;
                    this.leftWheel = -0.25;
                    return false;
                }
            }
            else if ((_state.ComLinkData.Yaw > -175) &&
(_state.ComLinkData.Yaw < 0))
            {
                if ((_state.ComLinkData.Yaw <= -170))
                {
                    this.rightWheel = -0.1;
                    this.leftWheel = 0.1;
                    // LogInfo(LogGroups.Console, String.Format("Yaw:
{0}", _state.ComLinkData.Yaw));
                    return false;
                }
                else
                {
                    this.rightWheel = -0.25;
                    this.leftWheel = 0.25;
                    return false;
                }
            }
            else
            {
                return true;
            }
        }


        /// <summary>
        /// Puts Robot looking doen ( Yaw in [-5,5]
        /// </summary>
        private bool lookDown()
        {
            //Rotation of the robot depends on its current orientation
            if ((_state.ComLinkData.Yaw < -5))
            {
                if ((_state.ComLinkData.Yaw > -10))
                {
                    this.rightWheel = 0.1;
                    this.leftWheel = -0.1;
                    return false;
                }
                else
                {
                    this.rightWheel = 0.25;
                    this.leftWheel = -0.25;
                    return false;
                }
            }
            else if (_state.ComLinkData.Yaw > 5)
            {
                if ((_state.ComLinkData.Yaw < 10))
                {
                    this.rightWheel = -0.1;
```

```
                this.leftWheel = 0.1;
                return false;
            }
            else
            {
                this.rightWheel = -0.25;
                this.leftWheel = 0.25;
                return false;
            }
        }
        else
        {
            return true;
        }
    }


    /// <summary>
    /// Checks whether the robot rotated since last loop
iteration.
    /// Returns true if it did not rotate, false elsewhere
    /// </summary>
    private bool noRotation()
    {
      if ((Math.Abs(prevYaw - _state.ComLinkData.Yaw) < 1))
        {
                    return true;
        }
        else
        {
            return false;
        }
    }

    /// <summary>
    /// Changes Robot State
    /// </summary>
    private void changeState(RoverState s)
    {
        prevState = _state.State;
        if (s == RoverState.RockFinder && rocksAnalyzed)
        {
            _state.State = RoverState.Navigate;
        }
        else
        {
            _state.State = s;
        }
    }

    /// <summary>
    /// Returns a list of locations that compose a path from a one
place to
    /// another
    /// </summary>
    private ArrayList getPath(comlink.Position from,
comlink.Position to)
    {
        ArrayList result = new ArrayList();
```

```
            //Every step will move (at most) ten meters on X and Z
axis

            int steps = (int)(Distance(from, to) / 10); //Number of
intermediate steps

            int xlength = (int) (Math.Abs(from.X - to.X));//Distance
between every step in X axis
            int zlength = (int)(Math.Abs(from.Z - to.Z));//Distance
between every step in Z axis
            bool fixedX = false;
            bool fixedZ = false;

            if (steps == 0)
            {
                result.Add(to);
                return result;
            }

            if (xlength < 5)
            { //If X distance is little, move directly to that
coordinate value
                fixedX = true;
            }

            if (zlength < 5)
            {//If Z distance is little, move directly to that
coordinate value
                fixedZ = true;
            }


            if (!fixedX)
            {
                xlength /= steps;

            }
            if (!fixedZ)
            {
                zlength /= steps;
            }


            //Calculate the position of every step and add it to the
path
            for (int i = 0; i < (steps-1); i++)
            {
                comlink.Position point = new comlink.Position();
                if (!fixedX)
                {
                    if (to.X > from.X)
                    {
                        point.X = from.X + (i + 1) * xlength;
                    }
                    else
                    {

                        point.X = from.X - (i + 1) * xlength;
                    }
```

```
                }
                else
                {
                    point.X = to.X;
                }

                if (!fixedZ)
                {
                    if (to.Z > from.Z)
                    {
                        point.Z = from.Z + (i + 1) * zlength;
                    }
                    else
                    {
                        point.Z = from.Z - (i + 1) * zlength;
                    }
                }
                else
                {
                    point.Z = to.Z;
                }
             result.Add(point);
            }
            comlink.Position finish = new comlink.Position();//Final
step: Destination
            finish.X = to.X;
            finish.Z = to.Z;
            result.Add(finish);
            return result;

        }


        /// <summary>
        ///  Calculates an alternative path to current one trying to
avoid an obstacle that
        ///  makes the robot get stuck.
        /// </summary>
        private ArrayList getPathAvoidStuck(comlink.Position from,
comlink.Position to)
        {
            ArrayList result = new ArrayList();
            ArrayList aux;
            ArrayList aux2;
             comlink.Position newPos = new comlink.Position();
            comlink.Position newPos2 = new comlink.Position();
            if (to.Z < from.Z)
            {
                newPos.Z = from.Z;
                newPos.X = from.X + 20;

                aux = getPath(from, newPos);
                for(int i = 0; i< aux.Count; i++){
                    result.Add(aux[i]);
                }
                newPos2.X = newPos.X;
                newPos2.Z = newPos.Z-20;
                aux2 = getPath(newPos, newPos2);
                for(int i = 0; i< aux2.Count; i++){

                    result.Add(aux2[i]);
```

```
        }

        aux = getPath(newPos, to);
        for(int i = 0; i< aux.Count; i++){
            result.Add(aux[i]);
        }
    }
    else if (to.Z > from.Z)
    {

        newPos.Z = from.Z;
        newPos.X = from.X - 20;

        aux = getPath(from, newPos);
        for(int i = 0; i< aux.Count; i++){
            result.Add(aux[i]);
        }
        newPos2.X = newPos.X;
        newPos2.Z = newPos.Z-20;
        aux2 = getPath(newPos, newPos2);
        for(int i = 0; i< aux2.Count; i++){

            result.Add(aux2[i]);
        }
        aux = getPath(newPos, to);
        for(int i = 0; i< aux.Count; i++){
            result.Add(aux[i]);
        }
    }else if (to.X > from.X)
    {

        newPos.X = from.X;
        newPos.Z = from.Z - 20;

        aux = getPath(from, newPos);
        for (int i = 0; i < aux.Count; i++)
        {
            result.Add(aux[i]);
        }
        newPos2.Z = newPos.Z;
        newPos2.X = newPos.X + 20;
        aux2 = getPath(newPos, newPos2);
        for (int i = 0; i < aux2.Count; i++)
        {

            result.Add(aux2[i]);
        }
        aux = getPath(newPos, to);
        for (int i = 0; i < aux.Count; i++)
        {
            result.Add(aux[i]);
        }
    }else if (to.X < from.X)
    {

        newPos.X = from.X;
        newPos.Z = from.Z + 20;

        aux = getPath(from, newPos);
        for (int i = 0; i < aux.Count; i++)
        {
```

```
                    result.Add(aux[i]);
                }
                newPos2.Z = newPos.Z;
                newPos2.X = newPos.X - 20;
                aux2 = getPath(newPos, newPos2);
                for (int i = 0; i < aux2.Count; i++)
                {

                    result.Add(aux2[i]);
                }
                aux = getPath(newPos, to);
                for (int i = 0; i < aux.Count; i++)
                {
                    result.Add(aux[i]);
                }
            }
            return result;

        }

        /// <summary>
        /// Auxiliar method that prints all the steps of a given path.
        /// Used for testing purpose
        /// </summary>
        private void printPath(ArrayList p)
        {
            foreach( comlink.Position pos in p){
                 LogInfo(LogGroups.Console, String.Format("STEP:
{0},{1}",pos.X,pos.Z));
            }
        }

        /// <summary>
        /// Performs a check on current path and modifies it if
necessary.
        /// </summary>
        private void pathRevision()
        {



            float distance = myDistance(_state.ComLinkData.Position,
target);
            int newPathNum = -1;
            pathRev = 0;
            for (int i = pathNum; i < path.Count; i++)
            {
                float distAux =
myDistance(_state.ComLinkData.Position, (comlink.Position)path[i]);
                if (distAux < distance)
                {
                    distance = distAux;
                    newPathNum = i;
                }
            }

            if (newPathNum > 0 && distance < 20)
            {
                //If current robot location is closer to a path step
that appears later in
                //its path, it heads towards it.
```

```
                //LogInfo(LogGroups.Console, String.Format("PATH
CHANGED DUE TO PROXIMITY"));
                pathNum = newPathNum;
                target = (comlink.Position)path[pathNum];

            }
            else if (distance >= 20)
            {
                //If robot distance to target is greater than 20
meters, it recalculates the
                //path to its destination

                //LogInfo(LogGroups.Console, String.Format("PATH
CHANGED DUE TO DISTANCE"));
                path = getPath(_state.ComLinkData.Position,
(comlink.Position)path[path.Count - 1]);
                pathNum = 0;
                target = (comlink.Position)path[pathNum];

            }


        }

        #endregion
    }
}
```