
**INGENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS**

Escuela Técnica Superior de Ingeniería Informática

Curso académico 2009-2010

**Proyecto Fin de Carrera**

# Soporte del robot humanoide Nao en simulador Gazebo para aplicaciones en JDERobot

**Tutor**: José María Cañas Plaza
**Autor**: Jorge Bermejo Juárez

*A mi familia,*
*por apoyarme siempre para que llegase hasta aquí.*
*Y a Cristina,*
*llegaste al final del camino, pero sin tí esto no tendría sentido*

# Agradecimientos

Quisiera dar las gracias a Jose María, tutor de este proyecto, por su apoyo, su tiempo, su dedicación y todos los conocimientos facilitados. Me gustaría extender este agradecimiento a todos los miembros del Grupo de Robótica de la Universidad Rey Juan Carlos por su predisposición a ayudar y coloborar.

Tambiém quisiera expresar mis más sinceros agradecimietos a mis compañeros de estudios por ofrecerme ayuda siempre que la he necesitado y con los que he compartido tantos momentos en esta Universidad. Gracias a ellos las largas jornadas se hacían algo más amenas.

Además, quisiera dar las gracias a mi familia, por darme todas las facilidades que han sido posibles para que consiguiese llegar a este momento, por su apoyo incondicional y por no dejar de creer nunca en mí. Y a Cristina, por aguantar las épocas de exámenes, los buenos y los malos momentos apoyandome siempre y creyendo y confiando en que podía conseguirlo.

*A todos: GRACIAS*

# Resumen

La utilización de simuladores es una práctica cada vez más habitual y extendida dentro de la infórmatica en general y de la robótica en particular. Este aumento en el uso de robots simulados viene determinado por una mejoría de los simuladores pues cada vez son más realistas, pudiendo simular mejor la física de los robots, siendo capaces de simular e incorporar cada vez más sensores e incluso muchos de los simuladores son capaces de simular varios robots al mismo tiempo y en 3 dimensiones.

Este proyecto de fin de carrera ha consistido en la creación y desarrollo del robot humanoide Nao de Aldebaran robotics en el entorno de simulación de robots Gazebo. Este robot es el utilizado en la liga de futbol robotico Robocup, competición en la que durante estos años ha venido participando la Universidad Rey Juan Carlos.

El objetivo final en la conseción del proyecto de fin de carrera es que el robot Nao simulado en Gazebo pueda ser utilizado con aplicaciones de JDERobot, muchas de las cuales estan diseñadas para que el robot Nao real realice algun tipo de comportamiento, como el de localizarse dentro del campo, etc.

Así, gracias a este proyecto, todas estas aplicaciones podrían ser probadas y mejoradas en el robot Nao simulado antes de ser pasadas al robot Nao real, obteniedo de este modo una ganancia en tiempo de desarrollo a tener muy en cuenta.

Para el desarrollo de la mayor parte del software creado en este proyecto de fin de carrera se han utilizado los lenguajes de programación C y C++, pues son los lenguajes utilizados tanto para el desarrollo del robot bajo la plataforma Gazebo, para la comunicación entre Gazebo y JDERobot, así como para las aplicaciones creadas con el fin de comprobar el correcto funcionamiento de nuestro robot Nao simulado.

# Índice general

# Índice de figuras

# Capítulo 1

# Introducción

En este capítulo se presentaran los elementos más importantes del contexto de nuestro proyecto, comenzando con una reseña historica de los inicios de la robótica. A continuación nos centramos en detallar un determinado tipo de robots, como son los robots humanoides cada vez mas comunes en nuestros días. También haremos una descripción detallada de la competición en la que está enfocado el proyecto, la RoboCup. Y finalizaremos con una descripción con ejemplos de simuladores de robots.

## 1.1. Robótica

De forma general, la robótica se define como el conjunto de conocimientos teóricos y prácticos que permiten concebir, realizar y automatizar sistemas basados en estructuras mecánicas poliarticuladas, dotados de un determinado grado de "inteligencia" y destinados a la producción industrial o a la sustitución del hombre en muy diversas tareas.

La Robótica es una tecnología multidisciplinar, pues hace uso de los recursos que le proporcionan otras ciencias, ya que en el proceso de diseño y construcción de un robot intervienen muchos campos pertenecientes a otras ramas de la ciencia, como por ejemplo la mecánica, la informática, la electrónica y la ingeniería artificial.

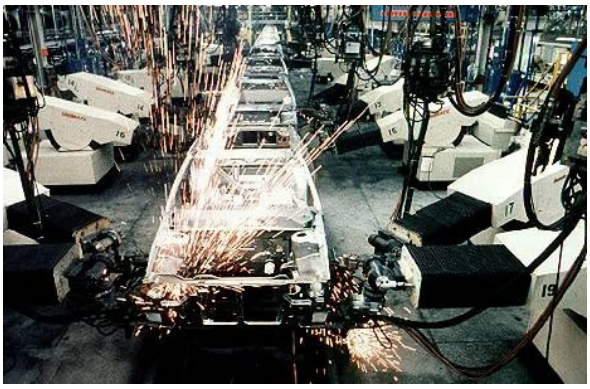
Podemos contemplar la robótica como una ciencia la cual, aunque se han conseguido grandes avances, todavía ofrece un amplio campo para el desarrollo y la innovación, y es precisamente este aspecto el que motiva a muchos investigadores y aficionados a los robots a seguir adelante planteando cada vez robots más evolucionados.

Los robots se componen esencialmente de tres tipos de dispositivos: sensores, procesadores y actuadores. En un robot los sensores son los encargados de recoger la información del entorno, estos dispositivos equivaldrían a nuestros sentidos humanos. En este grupo se situarían: el láser, el sonar o las cámaras. Por otro lado se encuentran los procesadores, encargados de analizar los datos que le son suministrados por los sensores. También son los encargados de elaborar una respuesta a estos datos, y

enviar la acción que deba llevarse a cabo a los actuadores. Estos últimos se encargan de interactuar con el entorno del mismo modo que lo hacen nuestros músculos.

El desarrollo tecnológico ha hecho posible la construcción de diferentes modelos robóticos capaces de ayudarnos o sustituirnos en la realización de tareas peligrosas, repetitivas, difíciles o que requieren una gran precisión. En sectores como la industria del automóvil la utilización de brazos robóticos (figura 1.1(a)) que sustituyen al hombre en las cadenas de montaje ha sido una constante desde que en 1967 se instalaran los primeros brazos robóticos en una factoría de General Motors. Hoy en día todas las empresas de la industria del automóvil utilizan esta tecnología en sus procesos de fabricación. En este ámbito los robot se encargan de labores de soldadura, pintado, ensamblaje de piezas, etc. (figura 1.1(b)) Pero los robots no

(a)



(b)

***Figura 1.1:*** *Brazo mecánico(a). Robots industriales fabricando coches (b)*

solo son utilizados en la industria del automóvil, sino que otras industria, como la alimenticia, tambien utilizan los robots para realizar sus propias tareas como el robot Adept Quattro(figura 1.2(a)) el cual puede moverse a 300 ciclos por segundo y que es utilizado para la manipulacion y empaquetamiento de alimentos. Desde entonces

(a)

(b)

***Figura 1.2:*** *Adept Quattro(a). Aspiradora Roomba (b)*

se han realizado grandes avances en este campo, existiendo actualmente robots para

todo tipo de propósitos como el rescate de personas y la localización de minas(figura 1.3(a)), salvando con ello muchas vidas humanas, además de utilizarse en diversos campos de la medicina, como la cirugía de alta precisión(figura 1.3(b)).
En un terreno un poco mas cercano como es el del hogar, tambien se han creado diversos robots para que nos ayuden a realizar las labores domésticas, en este apartado destaca actualmente el robot Roomba(figura 1.2(b)) una aspiradora que funciona sin cables y, además, de manera autónoma.

(a)

(b)

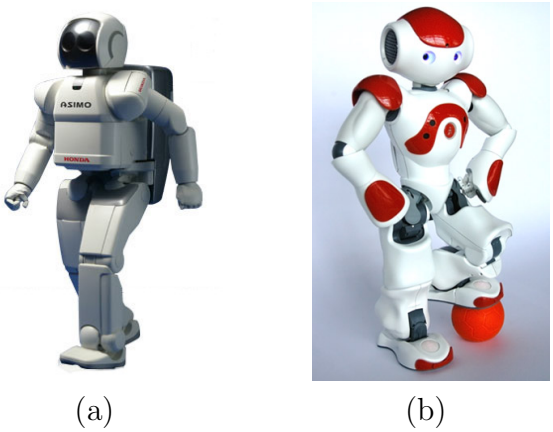
***Figura 1.3:*** *Robot desactivador de minas(a). Robot para cirugía Da Vinci (b)*

En la actualidad, existe una fuerte tendencia en ir aun más lejos. En este sentido, las grandes corporaciones japonesas como Toyota u Honda trabajan en el desarrollo de robots humanoides autónomos, con la intención de solventar un problema acuciante en la sociedad japonesa como es el rápido envejecimiento de la población y su falta de cuidado en el ámbito domestico.

El propio Bill Gates, responsable de la llegada de los ordenadores personales a los hogares de todo el mundo, afirmaba en el articulo “A robot in Every Home”, para la revista *Scientific American* [Gates, 2007]], que la situación de la robótica actual es muy similar a la del ordenador personal hace ya varios años, y que él confía en que a medio largo plazo empezaremos a tener robots en nuestras casas y a verlos progresivamente en las calles de nuestras ciudades.

## 1.2. Robots humanoides

Con el objetivo de favorecer la investigación en el campo de la robótica autónoma, diferentes empresas han desarrollado sus propios modelos de robot humanoides, el robot Asimo (figura 1.4(a)) , diseñado por Honda, y su homologo el prototipo diseñado por Toyota, cuentan con una gran agilidad motriz y utilizan sus cámaras como sensores principales, siendo el modelo de Honda capaz de reconocer y aprender nuevos objetos. Otro de los robots Humanoides mas destacados es el robot Nao de Aldebaran (figura 1.4(b)), el cual participa en la competición internacional de la robocup y del cual hablaremos mas adelante con mas detalle.




(a) (b)

***Figura 1.4:*** *Asimo(a). Nao (b)*

Un robot humanoide es un robot cuya apariencia general está basada en la del cuerpo humano, lo que le permite interactuar con herramientas y en entornos "para humanos". En general los robots humanoides estan constituidos por un torso, una cabeza, dos brazos y dos piernas, aunque algunos modelos de robots humanoides pueden estar construidos tan solo con alguna parte, como por ejemplo, solo de cintura para arriba.

Por definición, un robo humanoide es un robot autónomo ya que es capaz de adaptarse a los cambios del entorno o a los suyos propios para seguir con su meta. Esta es la principal diferencia entre los robots humanoides y cualquier otro tipo de robots.

Otra de las principales diferencias puede ser el aprendizaje autónomo, es decir, la capacidad de aprender o adquirir nuevas capacidades sin ayuda exterior.

Al igual que otros robots mecánicos los Humanoides también tienen los componentes básicos: sensores, procesadores y actuadores. Sin embargo, puesto que intentan simular la estructura y el comportamiento humano y son sistemas autónomos la mayoría de veces son más complejos que otro tipo de robots.

Esta complejidad afecta en a todas las escalas de la robótica, esto se traduce en que, por ejemplo, la mayoría de los humanoides actuales no son lo suficientemente fuertes como para poder saltar y esto sucede porque la relación potencia/peso no es tan buena como en el cuerpo humano. La complejidad llega incluso a la hora de desplazarse, pues los robots con ruedas son estables, mientras que hacer que robot bípedo ande sin caerse es bastante mas difícil.

Los robots humanoides son creados con el propósito de imitar algunas tareas físicas y mentales que los humanos desarrollan a diario. Científicos y especialistas de diferentes campos como la ingeniería, la ciencia cognitiva y la lingüística combinan sus esfuerzos para crear un robot lo más parecido a un humano posible. El objetivo es que algún día los robots sean capaces de comprender tanto la inteligencia como la razón humana y gracias a ello actuar como seres humanos. Además, gracias a

esa apariencia humana son más facilmente introducibles en los hogares, pues nos resultan mas familiares, más agradables de ver y menos "agresivos".

Uno de los beneficios del desarrollo de humanoides es entender la biología del cuerpo humano y los procesos mentales, desde el acto aparentemente simple de caminar a los conceptos de conciencia y espiritualidad. Los investigadores necesitan entender la estructura del cuerpo humano y su comportamiento (biomecánica) para la construcción y estudio de los robots humanoides.

Aunque el objetivo inicial de la investigación humanoide fue la de construir mejores órtesis y prótesis para los seres humanos como prótesis de la pierna con un deterioro neuromuscular, órtesis tobillo-pie, etc; los robots humanoides se están desarrollando para realizar tareas humanas como la asistencia personal, en las que deberían ser capaces de ayudar a las personas enfermas y a los ancianos, para las tareas del hogar (figura 1.5(a)) y para los trabajos insalubres o peligrosos (figura 1.5(b)). Puestos de trabajo ordinario como recepcionista o trabajador de una línea de fabricación de automóviles también son adecuados para los humanoides. En esencia, ya que podrán utilizar las herramientas y operar los equipos y el material diseñado para los humanos, los humanoides podrían, teóricamente, realizar las funciones que un ser humano puede realizar, siempre y cuando dispusiesen del software necesario, algo que no siempre es sencillo.

(a) (b)

***Figura 1.5:*** *Robots humanoides realizando tareas del hogar(a). Exhibición de robot humanoide realizando un experimento(b)*

## 1.3. Simuladores de Robots

Una herramienta muy útil en la programación de robots son los simuladores, los cuales ofrecen un entorno virtual en el que emulan las observaciones de los sensores y los efectos de las órdenes a los actuadores.

Pero, ¿Por qué no usar directamente robots reales? Hay distintos motivos para ello, como por ejemplo el tema económico. La utilización de simuladores evita la compra y el posterior mantenimiento del hardware del robot real, lo que abarata costes sobre todo cuando necesitarías contar con grupos numerosos de robots. Otra

ventaja es la facilidad de obtener ciertos datos, como la verdad absoluta sobre la posición del robot, más complejos o caros de obtener en el mundo real.

Otra de las ventajas principales es que permiten la evaluación, depuración y ajuste del programa de control antes de ser llevado al robot real. Esta es una ventaja muy interesante puesto que la repetitividad de los experimentos en los robots reales es mas compleja y requiere mas tiempo ya que las condiciones en un entorno real siempre cambian, como por ejemplo, la iluminación, el estado de las baterías, dónde se encuentre situado el robot, etc. Incluso, con los simuladores se puede arriesgar más a la hora de ajustar el programa ya que desaparece el miedo a que el robot se pueda dañar si se cae o si se choca.

A pesar de estas ventajas, el uso de simuladores no ha sido históricamente muy popular. Los primeros simuladores eran poco realistas y almacenaban mundos planos donde había obstáculos estáticos bidimensionales. Con los años se ha ido ganando en realismo, incorporando ruido en los sensores y en las actuaciones. Hoy en día se tienen simuladores tridimensionales, como Gazebo utilizado en este proyecto y el cual explicaremos con detalle en el tema 3, capaces de simular sensores tan complejos como la visión. En los simuladores más potentes se ha incluido también la capacidad de representar un conjunto de robots operando en el mismo escenario de modo simultáneo.

Muchos fabricantes incluyen un simulador(el cual es de pago) para sus robots como por ejemplo EyeSim para el robot EyeBot o Webots para Kephera y Koala, este último, además, incluye soporte para el robot Nao. Con Webots el usuario puede diseñar complejas configuraciones de robots, pudiendo poner en un mismo entorno compartido uno o varios, iguales o diferentes robots. Las propiedades de cada objeto, como la forma, el color, la textura, la masa, la fricción, etc, son elegidos por el usuario. Tambien dispone de una gran variedad de sensores y actuadores para dotar a cada robot. Los programas de control pueden ser transferidos a robots reales disponibles en el mercado.

También existen muchos desarrollos libres entre los que caben destacar:

*USARSim* (Sistema Unificado para la automatización y simulación de robots) es una simulación de alta fidelidad de los robots y ambientes basados en el motor del juego Unreal Tournament. Está concebida como una herramienta de investigación de propósito general con aplicaciones que van desde las interfaces hombre-máquina hasta la generación de comportamientos de grupos de robots heterogéneos. Además de usarse para aplicaciones de investigación, USARSim es la base para la competición RoboCupRescue con robots virtuales.(figura 1.6(a))

*Stage* simula numerosos robots móviles, sensores y objetos en un entorno de dos dimensiones.(figura 1.6(b)) Está diseñado para apoyar la investigación sobre sistemas multi-agentes autónomos, por lo que proporciona unos modelos bastante simples y de poco coste computacionalmente hablando en lugar de tratar de simular cualquier dispositivo con gran fidelidad. Este simulador pertenece al mismo proyecto que Gazebo.

(a) (b)

***Figura 1.6:*** *Simulación con USARSim(a). Robot Pioneer simulado con Stage(b)*

## 1.4. Robocup

La RoboCup [1] es un proyecto a nivel internacional para promover, a través de competencias integradas por robots autónomos, la investigación y educación sobre inteligencia artificial. Para ello se promueve un problema estándar donde un amplio abanico de tecnologías pueden ser integradas y examinadas. La meta final de la RoboCup es desarrollar un equipo de fútbol de robots humanoides totalmente autónomos capaces de jugar y ganar bajo normas de la FIFA contra el mejor equipo del mundo que exista en ese momento.

Para llegar a esta meta, se deben incorporar diversas tecnologías, a las que debe hacer frente cada robot, tales como razonamiento en tiempo real, utilización de estrategias, trabajo colaborativo entre robots, uso de múltiples sensores, autolocalización visual, etc.

La primera persona en pensar en robots que jugasen al fútbol fue Alan Maxworth en 1992, de la universidad de British Columbia de Canadá. Al año siguiente, y de forma independiente a la idea de Alan, se creó la primera competición de fútbol robótico en japón, llamada Robot J-League, pero viendo el interés internacional que surgió por este proyecto decidieron renombrarla a Robot World Cup Initiative o simplemente RoboCup. Finalmente en 1997, se realizó la primera competición de la RoboCup en el formato existente actualmente.

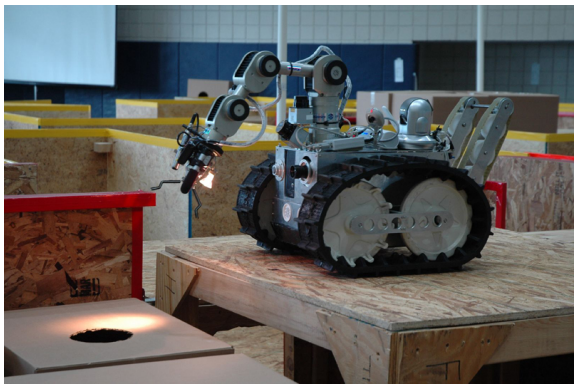
Las actividades realizadas por la RoboCup no son únicamente las competiciones entre robots, sino que también se componen de conferencias técnicas, programas educacionales, infraestructuras de desarrollo, etc.

A pesar de que inicialmente el fin de la RoboCup era el anteriormente explicado, se han creado nuevos campos de investigación relacionados con los robots y que forman parte del mismo proyecto, haciendo que la iniciativa se divida en cuatro

[1] http://www.robocup.org/

grandes competiciones:

- **RoboCupRescue:** En esta competición se pone a prueba a los robots en tareas de búsqueda y salvamento de víctimas en terrenos desfavorables. Los robots pueden ser tanto autónomos como guiados por control remoto (Figura 1.7(a)).
- **RoboCupJunior:** Trata de acercar las metas de la RoboCup a estudiantes de educación primaria y secundaria.
- **RoboCup@Home:** Centrada en la utilización de robots autónomos para realizar tareas del hogar y la vida diaria (Figura 1.7(b)).
- **RoboCupSoccer:** Es el proyecto principal de la Robocup y donde esta centrado nuestro proyecto por lo que es la competición que vamos a explicar con más detalle.



(a)

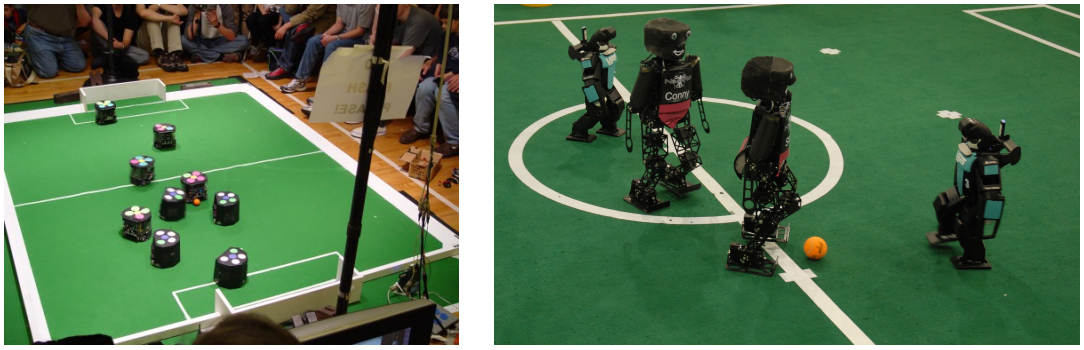
(b)

***Figura 1.7:*** *Robot de la RoboCupRescue(a). Robot de la RoboCup@Home (b)*

Dentro de esta competición existen a su vez varias subcompeticiones:

- *Liga de simulación:* No existen robots físicos, por lo que sólo se enfrentan robots en simulaciones virtuales. Cada robot simulado puede tener sus propias características.
- *Liga de robots de tamaño pequeño:* Categoría centrada en la cooperación multiagente, donde sólo pueden utilizarse robots de menos de 15 cm de altura, con un diámetro menor de 18 cm (Figura 1.8(a)) y que cuentan con una cámara cenital como sensor .
- *Liga de robots de tamaño mediano:* Se utilizan robots con una altura desde los 30 a los 80 cm y con un peso máximo de 40 kg. Los robots utilizados deben ser totalmente autónomos y pueden comunicarse entre ellos mediante wifi, siendo la visión su sensor principal.

- *Liga de humanoides:* Centrada en el desarrollo de robots humanoides, por lo tanto, se hace más hincapié en el hardware utilizado que en el comportamiento software (Figura 1.8(b)).



(a) (b)

***Figura 1.8:*** *Partido de la liga de tamaño pequeño(a). Partido de la liga de humanoides (b)*

- *Plataforma estándar:* Categoría en la que todos los participantes utilizan el mismo tipo de robot y sólo se centran en el desarrollo del software de éste. Actualmente se utiliza el robot Nao de Aldebaran Robotics, aunque hasta 2007 se utilizó el robot Aibo de Sony. Los robots utilizados deben ser totalmente autónomos y deben utilizar una cámara como sensor principal. Esta liga, junto con la liga de humanoides, es una de las que más ha progresado y la más cercana al reto fijado.

Además de la competición oficial de la RoboCup celebrada anualmente, también se han creado otros torneos a nivel mundial en los que poder competir con los robots, así, ya dentro de la plataforma estándar, se han creado nuevas competiciones que siguen las mismas reglas que la RoboCup, como el German Open, el Japan Open o el Mediterranean Open.

Dentro de todas las competiciones de la RoboCup, la que a nosotros más nos interesa es la plataforma estándar, pues el modelo de robot usado para participar en esta competición, el robot Nao de Aldebaran Robotics, es el que vamos a modelar y desarrollar en el entorno de simulación de robots Gazebo.

En el siguiente capítulo, desarrollaremos los objetivos concretos que pretendemos cumplir en este proyecto fin de carrera, así como los requisitos subyacentes a estos. A continuación, en el tema 3 detallaremos tanto la plataforma como las herramientas empleadas para poder satisfacer los objetivos marcados. El detalle en profundidad del software diseñado y programado lo daremos en el tema 4, analizando en el tema 5 las aplicaciones que hemos desarrollado para probar el correcto funcionamiento de nuestro proyecto. Por último, en el tema 6 haremos alusión a los resultados obtenidos y a los posibles trabajos futuros derivados de nuestro proyecto.

# Capítulo 2

# Objetivos y metodología

Una vez presentado el contexto general en el que se desarrolla nuestro trabajo, en este capítulo vamos a detallar los objetivos concretos que pretendemos alcanzar con nuestro proyecto, así como los requisitos que debe cumplir nuestra solución.

## 2.1. Descripción del problema

El objetivo general de este proyecto es crear un robot Humanoide dentro de un entorno simulado. Este robot simulado tendrá las mismas características que su homólogo real, tanto en tamaños, pesos, número de articulaciones y grados de libertad como en los sensores que incorpora.

Este robot Humanoide simulado nos permitirá realizar pruebas de aplicaciones antes de pasarlas al robot real. Esto nos supondrá un ahorro en tiempo, pues podremos cambiar alguna parte del código y probarlo de forma más rapida que si le tuviesemos que pasar dicho código al robot real. Además, evitará daños en el robot real.

Este objetivo general lo hemos articulado en tres subobjetivos que nos permitirán alcanzar el objetivo principal, y que se describen a continuación:

1. Creamos el robot dentro del entorno de simulación. Este robot estará construido a base de figuras geometricas básicas pero dándole ya una apariencia de humanoide. El tamaño de este robot humanoide simulado, así como sus masas serán iguales que las del robot real. Además, le dotaremos de articulaciones y grados de libertad basandonos también en dicho robot real. Dentro de este subobjetivo también está el añadirle alguno de los sensores que incorpora el robot real, como por ejemplo una cámara.

2. Dotaremos al robot simulado de una apariencia lo más cercana posible a la del robot real. Para ello moldearemos con algun programa de diseño 3D la piel del robot.

Esta piel será incorporada al modelo creado ateriormente así obtendremos un robot simulado mas "real", no como se vería creados tan solo con figuras geométricas básicas.

3. Desarrollaremos los componentes necesarios para dar soporte al control y manejo del robot humanoide simulado. De esta forma podremos controlar sus articulaciones y dotarlas de movimiento, así como acceder al sensor de la cámara y obtener sus imágenes.

## 2.2. Requisitos

Teniendo en cuenta los objetivos de la sección anterior, el proyecto deberá satisfacer una serie de requisitos descritos a continuación:

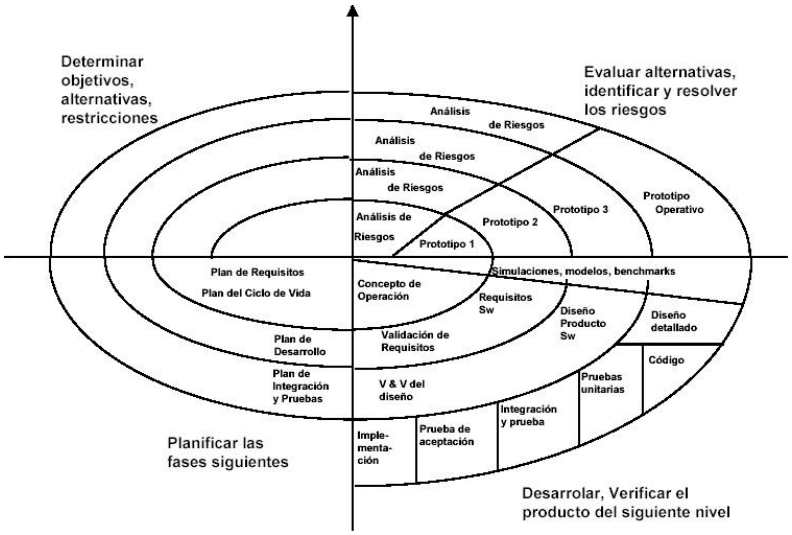
- El robot humanoide a simular será el empleado para la liga estándar de la robocup, el robot Nao de Aldebaran Robotics, el cual detallaremos en el capítulo 3.
- El robot simulado será desarrollado para la plataforma de simulación de robots Gazebo en su versión 0.7. Este simulador 3D está explicado con más detalle en el capítulo 3.
- Para la creacion de la apariencia externa del robot se usará el programa de diseño 3D Blender. Nos basaremos en el robot Nao real para construir cada una de las partes del robot intentando que sean lo más parecidas en apariencia.
- Este proyecto hará uso de la arquitectura de desarrollo JDERobot, con la que trabaja el grupo de robótica de la URJC. Esta arquitectura está desarrollada sobre el lenguaje de programación C y C++.

## 2.3. Metodología de desarrollo

En el desarrollo del sistema descrito, el modelo de ciclo de vida utilizado ha sido el modelo en espiral basado en prototipos, ya que permite desarrollar el proyecto de forma incremental, aumentando la complejidad progresivamente y haciendo posible la generación de prototipos funcionales.

Este tipo de modelo de ciclo de vida nos permite obtener productos parciales que puedan ser evaluados, ya sea total o parcialmente, y facilita la adaptación a los cambios en los requisitos, algo que sucede muy habitualmente en los proyectos de investigación.

El modelo en espiral se realiza por ciclos, donde cada ciclo representa una fase del proyecto. Dentro de cada ciclo del modelo en espiral se pueden diferenciar 4 partes principales que pueden verse en la figura 2.1(a), y donde cada una de las partes tiene un objetivo distinto:




(a)

***Figura 2.1:*** *Modelo de desarrollo en espiral basado en prototipos (a).*

- Determinar objetivos: Se establecen las necesidades que debe cumplir el sistema en cada iteración teniendo en cuenta los objetivos finales, por lo que según avancen las iteraciones aumentará el coste del ciclo y su complejidad.
- Evaluar alternativas: Determina las diferentes formas de alcanzar los objetivos que se han establecido en la fase anterior, utilizando distintos puntos de vista, como el rendimiento que pueda tener en espacio y tiempo, las formas de gestionar el sistema, etc. Además se consideran explícitamente los riesgos, intentando mitigarlos lo máximo posible.
- Desarrollar y verificar: Desarrollamos el producto siguiendo la mejor alternativa para poder alcanzar los objetivos del ciclo. Una vez diseñado e implementado el producto, se realizan las pruebas necesarias para comprobar su funcionamiento.
- Planificar: Teniendo en cuenta el funcionamiento conseguido por medio de las pruebas realizadas, se planifica la siguiente iteración revisando posibles errores cometidos a lo largo del ciclo y se comienza un nuevo ciclo de la espiral.

Los ciclos que se han seguido en nuestro proyecto están relacionados con cada una de las etapas que se describirán en la siguiente sección. A lo largo de estas etapas, se han realizado reuniones semanales con el tutor del proyecto para negociar los objetivos que se pretendían alcanzar y para evaluar las alternativas de desarrollo.

## 2.4. Plan de trabajo

Atendiendo a la metodología seguida, durante el transcurso del proyecto se han marcado progresivamente una serie de requisitos a cumplir dividiendo el trabajo a

realizar en distintas etapas:

1. **Familiarización con el simulador Gazebo:** En esta fase trataremos de familiarizarnos con el simulador de robots 3D Gazebo. Para ello, a parte de usar los modelos de robots ya creados desarrollaremos los nuestros propios basandonos en alguno ya existente. Tambien en esta fase incorporaremos dicho modelo creado por nosotros mismos en otro modelo precreado. Así, adquiriremos los conocimientos necesarios tanto para crear modelos como para ir incorporando unos en otros.

2. **Construcción modelo Humanoide:** Una vez que ya nos hemos familiarizado con el simulador Gazebo pasamos a construir un modelo de robot con forma humanoide. Empezamos por construir un modelos sin piernas, solo con el tronco, la cabeza y los brazos con el fin de que nuestro modelo sea estable y no se caiga facilmente. Más tarde pasaremos a construirle unas piernas e incorporarlas a nuestro modelo anterior.

3. **Robot Nao simulado:** En esta etapa ya tenemos construido nuestro modelo de robot humanoide en gazebo. El siguiente paso es hacer que éste se parezca y tenga las mismas características que el Robot Nao de Aldebaran Robotics. Para ello ajustamos las medidas, masas, articulaciones, sensores, etc de nuestro modelo humaniode a las del robot Nao real.

4. **Familiarización con JDERobot:** El objetivo de esta fase es aprender a utilizar el software JdeRobot y las aplicaciones y drivers que contien, ya que algunos de estos formarán parte del proyecto final. Se hace uso del componente creado por Francisco Rivas NaoOperator, con el cual podemos acceder a todas las articulaciones del robot Nao. Además, tambien en esta etapa crearemos nuestro propio driver para poder mover las articulaciones de nuestro robot Nao simulado.

5. **Soporte Nao simulado en driver Gazebo:** En esta etapa incorporaremos al driver que utiliza JdeRobot para comunicarse con Gazebo el soporte para el control y manejo de un robot Humanoide. En nuestro caso del robot humanoide Nao.

6. **Pruebas de uso:** Con el fin de comprobar que todo funciona correctamente creamos 2 componentes para JdeRobot. Estos componentes prueban por un lado el correcto funcionamiento de uno de los sensores que incorpora el robot Nao, como es la cámara de la cabeza y por otro el correcto funcionamiento de todas y cada una de las articulaciones. Para realizarlo creamos un componente llamado **faceball** el cual busca la pelota dentro del campo con la cámara y una vez que la localiza la sigue de tal forma que la pelota siempre esté en el centro de la visión.

   El otro componente lo hemos llamado **dance** y su objetivo es que el robot Nao simulado en Gazebo mueva todas las articulaciones de su cuerpo mediante la realización de un baile sencillo.

# Capítulo 3

# Entorno de trabajo

En este capítulo vamos a describir los elementos empleados en el desarrollo del proyecto, tanto hardware como software.

## 3.1. Gazebo

Gazebo[1] es un simulador 3D que ofrece un entorno rico para desarrollar y probar de forma rápida sistemas multi-robots. Es una herramienta eficaz, escalable, sencilla y forma parte del proyecto Player/Stage[2] por lo que es software libre bajo licencia GPL. Actualmente Gazebo se encuentra en su version 0.8, aunque para la realización de este proyecto hemos utilizado la versión 0.7.
Está diseñado para reproducir con precisión los entornos dinámicos que un robot pueda encontrar. Todos los objetos simulados tienen masa, velocidad, fricción, y otros numerosos atributos que les permiten comportarse de forma realista.
Los robots son estructuras dinámicas compuestas por cuerpos rígidos conectados a través de articulaciones. A la superficie de estas articulaciones se le pueden aplicar fuerzas, tanto angulares como lineales, para generar locomoción e interacción con el entorno.
Una de las principales características de Gazebo es que además de poder simular robots, sensores, actuadores, etc. nos permite la posibilidad de crear los nuestros propios fácilmente.
La arquitectura software se ilustra gráficamente en la figura 3.1(a). El "Mundo", que es nuestro escenario a simular, representa el conjunto de todos los modelos y factores ambientales tales como la gravedad y la iluminación. Cada modelo se compone de al menos un cuerpo y cualquier número de articulaciones y sensores. Las librerías de terceros se comunican con Gazebo al nivel más bajo. Esto evita que los modelos se conviertan en dependientes de herramientas específicas que pueden cambiar en el futuro. Por último, los comandos del cliente y los datos se reciben o son devueltos, respectivamente, a través de una interfaz de memoria compartida para, de este modo, obtener una mayor rapidez de acceso. Un modelo puede tener muchas interfaces para funciones que impliquen, por ejemplo, el control de las articulaciones

[1] http://playerstage.sourceforge.net/gazebo/gazebo.html
[2] http://playerstage.sourceforge.net/

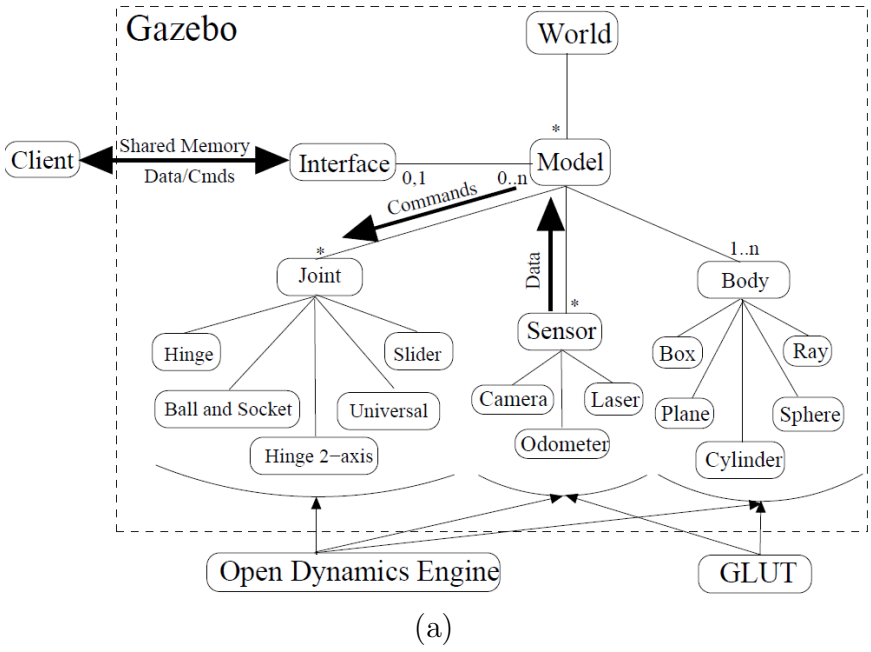


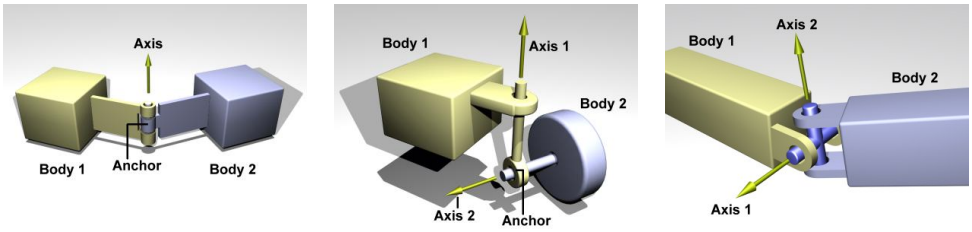

***Figura 3.1:*** *Estructura general de la arquitectura Gazebo*

y la transmisión de imágenes de la cámara.

En cuanto a la simulación de la física utiliza ODE[3] (*Open Dynamics Engine*), uno de los motores físicos más utilizados en la comunidad de código abierto al ser, al igual que Gazebo, software libre. Un motor físico lo que nos permite es simular masas, rozamientos, inercias, choques, dinámicas, etc. En concreto, ODE está diseñado para simular la dinámica y la cinemática asociada a los cuerpos rígidos articulados. Este motor incluye muchas características tales como numerosas articulaciones (figura 3.2(a)), detección de colisiones, la masa y las funciones de rotación, y muchas geometrías(figura 3.3(a)).
Gazebo utiliza estas características proporcionando una capa de abstracción situada entre ODE y los modelos. Dicha capa permite la fácil creación de normales y objetos abstractos tales como los rayos láser, mientras se mantiene toda la funcionalidad proporcionada por ODE.

Un simulador bien diseñado suele proporcionar algún tipo de interfaz el cual permite al usuario humano ver lo que está pasando en el mundo simulado. Para realizar esta labor Gazebo requiere de un interfaz que sea a la vez sofisticado y rápido. El corazón de Gazebo reside en su capacidad para simular la dinámica, y esto requiere una importante labor en nombre de la computadora del usuario. Un lento y engorroso interfaz de usuario sólo iría en detrimento del propósito principal del simulador. Para evitar esto, OpenGL y GLUT (*OpenGL Utility Toolkit*) fueron elegidos como las herramientas de visualización por defecto.
OpenGL es una biblioteca estándar para la creación de aplicaciones 2D y 3D

[3]http://ode.org/




(a)

***Figura 3.2:*** *Diferentes modelos de articulaciones*

(a)

***Figura 3.3:*** *Diferentes modelos de geometrías*

interactivo. Es independiente de la plataforma, altamente escalable, estable y está en constante evolución. Más importante aún, muchas características en OpenGL se han implementado en hardware de la tarjeta gráfica lo que libera a la CPU para otras tareas.
Por su parte GLUT es un sistema de ventanas simple con un conjunto de herramientas independientes para las aplicaciones OpenGL. Escenas renderizadas utilizando OpenGL se muestran en las ventanas creadas por GLUT. Esta herramienta también proporciona mecanismos para la interacción del usuario con Gazebo a través de dispositivos de entrada estándar como teclados y ratones. Con estas herramientas podemos generar la simulación de los modelos de cámaras que por defecto incorpora Gazebo, como por ejemplo la SonyVID30, asi como la imágen de la cámara de observación para ver el escenario simulado.

Este escenario es esencialmente una colección de modelos y sensores. Un modelo es cualquier objeto que mantiene una representación física. Esto abarca todo, desde la geometría simple hasta robots complejos. Los modelos están formados por al menos un cuerpo rígido, cero o más articulaciones y sensores e interfaces para facilitar el flujo de datos.

Los Cuerpos representan los bloques básicos de construcción de un modelo. Su representación física es en forma de figuras geométricas básicas como cajas, esferas, cilindros, planos y líneas. A cada cuerpo se le asigna una masa, la fricción, el factor de rebote, y renderizado en propiedades como el color, textura, transparencia, etc. Las articulaciones proporcionan el mecanismo para conectar cuerpos entre sí con el fin de formar relaciones cinemáticas y dinámicas. Además de la conexión de dos cuerpos, estas articulaciones pueden actuar como motores. Para ello se aplica una fuerza a una articulación, y la fricción entre los cuerpos conectados produce el movimiento. No

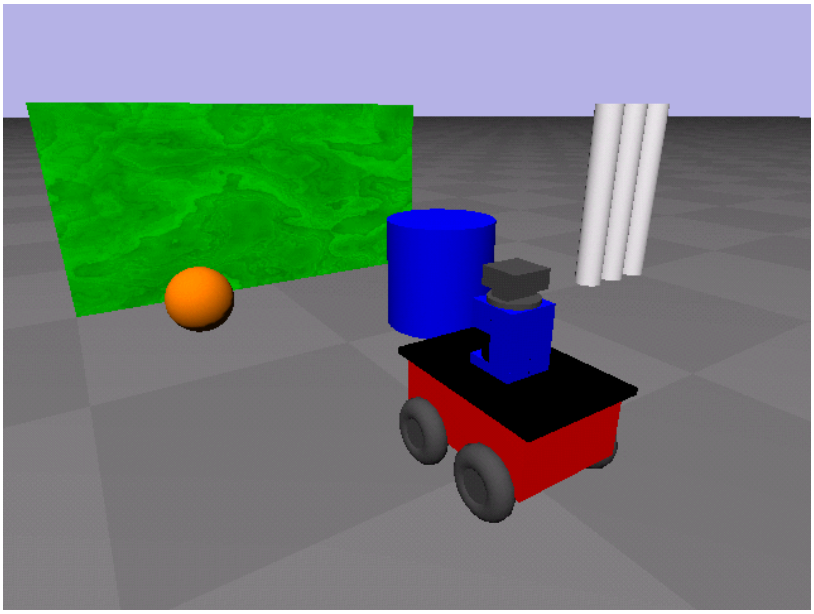


***Figura 3.4:*** *Mundo en Gazebo*

obstante, hay que tener especial atención cuando se conectan muchas articulaciones en un modelo único, pues tanto el modelo como la simulación pueden fácilmente perder la estabilidad si los parámetros elegidos no son los correctos. Hay disponible una gran variedad de articulaciones, como se puede ver en la figura(3.2(a)).

Un sensor en Gazebo es un dispositivo abstracto que carece de una representación física. Sólo sirven cuando se incorporan a un modelo. Esta característica permite la reutilización de los sensores en numerosos modelos reduciendo así el código y la confusión. Los sensores permanecen separados de la simulación dinámica, ya que sólo recogen datos, o los emiten si se trata de un sensor activo.

El terreno y los edificios representan modelos inmóviles mientras los robots y otros objetos son dinámicos. En Gazebo hay varios modelos de robots ya creados como por ejemplo el Pioneer2DX, Pioneer2AT, SegwayRMP, asi como también existen varios sensores como por ejemplo el LaserSick, la cámara SonyVID30, etc. Todos estos sensores y modelos pueden ser utilizados para la creación de otros modelos de robots.

En la actualidad los modelos son creados a mano. El proceso se inicia con la elección de un cuerpo apropiado y las articulaciones necesarias para construir un modelo preciso, tanto en apariencia como en funcionalidad. El siguiente paso es unir los cuerpos entre sí usando las articulaciones. El resultado final es una representación física completa de nuestro modelo. Una vez hecho esto a nuestro modelo se le pueden incorporar cualquier número de dispositivos como cámaras, sensores odometricos, etc.(3.5(a))

Una vez que hemos creado el modelo, con todas las articulaciones y sensores que hayamos querido poner es hora de darle movimiento al robot. Para ello necesitamos un programa cliente que se comunique con Gazebo. Existen dos formas para este proposito, una mediante Player y la otra mediante libgazebo. Nosotros nos hemos

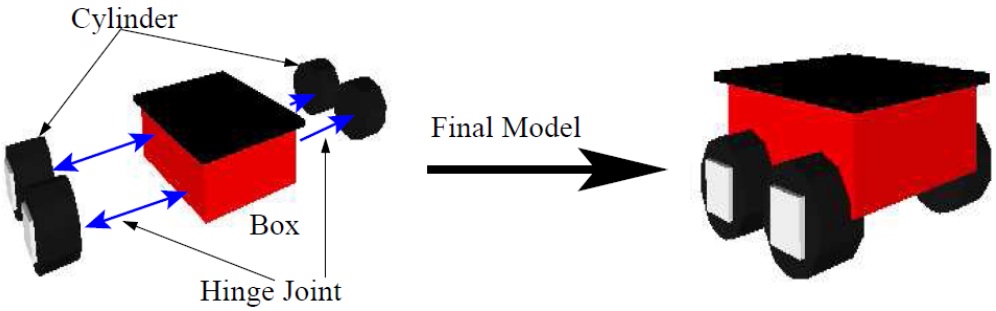



***Figura 3.5:*** *Construcción del Pioneer2 AT*

decidido a utilizar esta última pues aunque requiere algo más de trabajo a la hora de programar es más rápida ya que tenemos un interfaz directo entre nuestro programa cliente y Gazebo.

Siguiendo estos pasos hemos diseñado el Robot Nao de Aldebaran en un entorno simulado, uno de nuestros propositos en la realización de este proyecto.

## 3.2. JdeRobot

Esta plataforma software ha sido creada por el grupo de robótica de la URJC, se trata de una plataforma para aplicaciones con robots y visión artificial. JdeRobot está desarrollado en C y C++ y proporciona un entorno de programación donde la aplicación se compone de distintos hilos de ejecución asíncronos llamados esquemas. Cada esquema es un *plugin* que se carga automáticamente en la aplicación y que realiza una funcionalidad especifica que podría ser reutilizada.

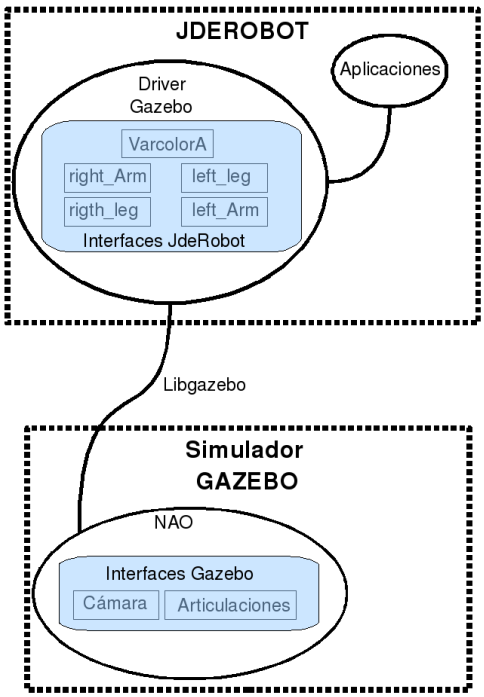
Existen dos tipos de esquemas en JdeRobot, los perceptivos, que son los encargados de realizar algún tipo de procesamiento de datos para proporcionar información sobre el robot y el mundo en el que opera, y los esquemas de actuación, que se encargan de tomar decisiones para alcanzar una meta, como lanzar ordenes a los motores o lanzar nuevos esquemas, ya que pueden combinarse formando jerarquías.

JdeRobot simplifica el acceso a los dispositivos hardware desde el programa, de modo que leer la medida de un sensor es simplemente leer una variable local y lanzar órdenes a los motores es tan simple como escribir en otra. Para conectar los esquemas de la aplicación con los sensores y actuadores del robot existen un conjunto de drivers, que también actúan como *plugins*, y son los encargados de dialogar con los dispositivos hardware concretos.

El driver que hemos utilizado en nuestro proyecto ha sido el driver gazebo el cual permite la conexión de una aplicación de JDERobot con el simulador Gazebo. El simulador Gazebo ofrece gran cantidad de sensores y actuadores. Por ejemplo, el driver permite la recepción de imágenes generadas por cámaras simuladas, dar órdenes de movimiento al robot, o controlar el cuello mecánico de una cámara.

Además, ofrece de forma transparente, mediante interfaces, la información que se transmite desde el simulador, y traduce la información generada por nuestra aplicación para que podamos dar ordenes al robot.

En nuestro caso, el driver gazebo nos ofrece a través de varcolorA la imágen captada por la cámara, a través de ptencoders el estado del cuello y a través de left_arm, right_arm, left_leg y right_leg la posición de las articulaciones de los brazos y de las piernas, a través de estos últimos también se pueden mandar órdenes para poner dichas articulaciones en una determinada posición. En la figura (3.6(a)) podemos apreciar como se realiza la comunicación entre los diferentes componentes y su jerarquía.




(a)

***Figura 3.6:*** *Esquema(a).*

La principal razón por la que se ha elegido JdeRobot es que ofrece una interfaz sencilla para la programación de sistemas de tiempo real y resuelve problemas relacionados con la sincronización de los procesos y la adquisición de datos. Además esta preparada para aprovechar la potencia de los procesadores multinúcleo, evitando por ejemplo que la interfaz de usuario ralentice el procesamiento de la información. Otras razones son la facilidad con la que se pueden utilizar drivers y aplicaciones ya creados para esta plataforma de desarrollo y la posibilidad de reutilizar código de proyectos con partes similares.

El driver gazebo incluido en la versión 4.3 de JDERobot no soporta el control de un robot humanoide, por lo que fue necesario desarrollar en este proyecto un nuevo driver. El nuevo driver Gazebo desarrollado se ha incorporado a la versión oficial de

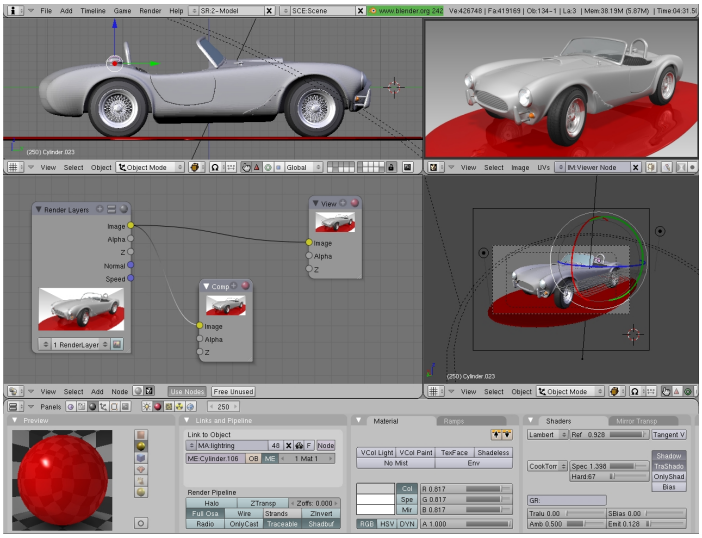
JDErobot y constituye un aporte lateral de este proyecto a la infraestructura de la plataforma.

## 3.3. Blender

Blender[4] es una aplicación de gráficos 3D. Esta aplicación permite modelar, dar textura, animar, renderizar, simular efectos de partículas, y hasta editar y sincronizar audio y video en forma no-lineal o incluso crear aplicaciones 3D como videojuegos. Además, también nos ofrece herramientas de simulación avanzada tales como dinámicas de cuerpos rígidos, de cuerpos suaves y de fluidos, herramientas de animación de personajes, un sistema de simulación física avanzado, y un sistema de composición de materiales basado en nodos.

Por el contrario tiene una interfaz gráfica de usuario poco amigable y criticada en muchos casos como poco intuitiva, pues no se basa en el sistema clásico de ventanas. Para muchos, más que un inconveniente esto supone una ventaja pues te permite una configuración personalizada de la distribución de los menús y vistas de cámaras(figura 3.7(a)).

Blender es software multiplataforma, libre y gratuito, la versión usada en este



(a)

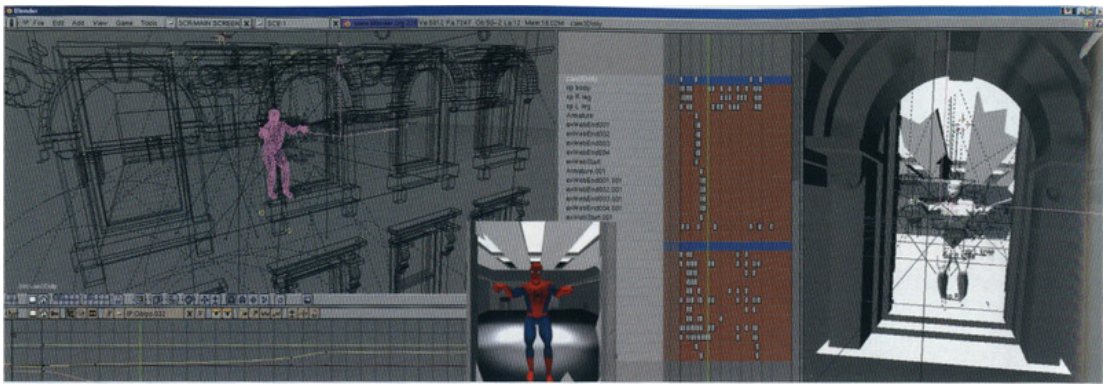
***Figura 3.7:*** *Ejemplo de interfaz de Blender.*

proyecto y la última disponible es la 2.49. Tiene capacidad para crear una gran variedad de primitivas geométricas, incluyendo curvas, mallas poligonales, vacíos, NURBS, metaballs, así como de un sistema de partículas estáticas para simular cabellos y pelajes, al que se han agregado nuevas propiedades entre las opciones de shaders para lograr texturas realistas.

[4]http://www.blender.org/

Dispone también de posibilidades de renderizado interno versátil e integración externa con potentes trazadores de rayos o raytracer"libres como kerkythea, YafRay o Yafrid. Esta aplicación ha sido utilizada en diversos cortos de animación, algunos de ellos hechos exclusivamente con Blender, y en películas como Spiderman 2, en donde fue utilizado para crear previsualizaciones del storyboard (figura 3.8(a)).

Blender ha sido utilizado en este proyecto para crear la estructura externa del robot Nao, la piel. Puesto que Gazebo solo permite la construcción de modelos mediante figuras geometricas básicas era necesario cargar estas pieles para darle al robot Nao simulado una apariencia mas parecida al robot real.



(a)

***Figura 3.8:*** *StoryBoard de la pelicula SpiderMan 2.*

## 3.4. Nao

El robot Nao(figura 3.9(a)) es el robot utilizado en la liga de la plataforma estándar de la RoboCup desde el año 2008, donde sustituyó al robot Aibo de Sony(figura 3.9(b)). Se trata de un robot humanoide desarrollado por Aldebaran Robotics[5].

Este robot humanoide mide 58 centímetros de alto y pesa unos 4,3 kilogramos. Dispone de varios sensores como los FSR o detectores de presión en los pies, un sensor inercial, 4 sonars en el pecho, los cuales permiten al robot estimar la distancia a los obstáculos que hay en su entorno en rango de detección de 0 a 70 centímetros, aunque por debajo de los 15 centímetros no es capaz de calcular la distancia exacta sino tan solo saber que hay un objeto presente. Incorpora, además, 2 cámaras con distintas zonas de visión en su cabeza(figura 3.10(b)). Estas cámaras nos proporcionan imágenes de una resolución de 640x480 píxeles a 30 fps. y permiten al robot Nao capturar fotos, secuencias de video, reconocer objetos de colores, detectar y reconocer caras, etc.

[5]http://www2.aldebaran-robotics.com/en

(a) (b)

***Figura 3.9:*** *Robot Nao con la pelota de la RoboCup(a). Aibo de Sony(b).*

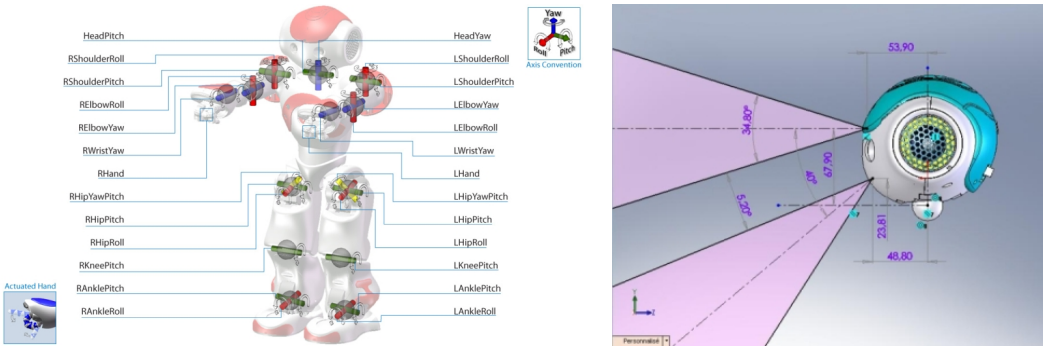
Consta también de 2 altavoces y varios micrófonos e integra un sistema de reconocimiento de voz que localiza de dónde viene el sonido de modo que pueda girar su cabeza hacia el origen de ese sonido y un sistema de reproducción de texto a voz para que pueda interactuar con su entorno mediante la pronunciación verbal de cualquier texto que le haya sido enviado.

El robot Nao de Aldebaran es totalmente autónomo, es capaz de encontrar un camino alrededor de una habitación llena de obstáculos e, incluso, ir solo a su base de recarga cuando su batería esta demasiado baja. Es capaz de realizar una gran amplitud de movimientos como caminar, sentarse, levantarse, bailar, evitar obstáculos, coger objetos, etc. Para poder realizar todos estos movimientos el robot Nao dispone de 25 grados de libertad (figura 3.10(a)) divididos de la siguiente forma

- 2 Grados de libertad en la cabeza.
- 5 Grados de libertad en cada brazo.
- 1 Grado de libertad en cada mano.
- 1 Grado de libertad en la pelvis.
- 5 Grados de libertad en las piernas.

Está equipado de serie con un procesador x86 AMD GEODE 500MHz, usa como sistema operativo Linux y permite comunicación vía Ethernet o Wi-fi.

Para fines de entretenimiento, Nao puede interactuar con su dueño, gracias a la evolución de sus comportamientos y funcionalidades. Está siendo introducido poco a poco en este mercado con el fin de ser un compañero más de la familia y, en un futuro, con funciones más sofisticadas adaptar un nuevo rol, el de ayudar a los seres humanos con las tareas diarias.




(a) (b)

***Figura 3.10:*** *Grados de libertad del robot Nao(a). Cámaras del robot Nao(b).*

El software de creación de movimientos *Choregraphe* está diseñado para ajustarse al nivel de programación de cualquier usuario, desde los más principiantes a los mas avanzados. Así, el robot humanoide Nao es un compañero ideal para la investigación y la docencia en el campo de la robótica y la inteligencia artificial.

Todo ello convierte al robot Nao en la plataforma de desarrollo de robots humanoides más competente hoy en día.

Este robot Humanoide ha servido como modelo para nuestro proyecto ya que nos hemos basado en él, en sus medidas, masas, sensores, etc. para construir nuestro modelo de robot Humanoide simulado.

# Capítulo 4

# Soporte del Nao simulado

Una vez explicados los requisitos y herramientas utilizadas para el desarrollo del proyecto, explicaremos en profundidad el software diseñado y programado. Para ello vamos a comenzar dando una visión general del diseño software y posteriormente se describirán sus detalles y la implementación que se ha llevado a cabo en cada una de sus partes.

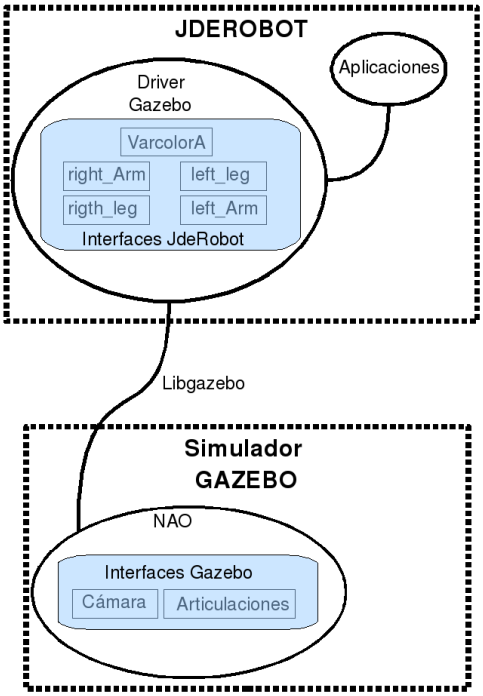
## 4.1. Diseño general

El objetivo global del proyecto es diseñar y desarrollar soporte para el robot Nao de Aldebaran en el entorno simulado Gazebo y utilizarlo en aplicaciones de JDERobot. Actualmente Gazebo no da soporte a ningún tipo de robot bípedo con lo que nuestro primer objetivo fue el de diseñar un robot con forma Humanoide que se mantuviese estable sobre dos piernas. Gazebo nos ofrece mecanismos para incorporar nuevos robots, construyéndolos a partir de objetos compuestos por bloques básicos ya definidos. Este robot humanoide se crea como un *plugin* para el simulador Gazebo. Este plugin, al igual que el de modelos ya creados, posee una serie de interfaces software las cuales nos ofrecen datos referentes al robot, como las imágenes de la cámara o la posición de las articulaciones, y con las cuales podemos ejercer la comunicación con la plataforma de desarrollo de aplicaciones robóticas JDERobot.

La comunicación con las aplicaciones de JDERobot se realiza gracias al *driver* Gazebo, este *driver* nos proporciona la intercomunicación entre el simulador Gazebo y JDERobot. El *driver* Gazebo ofrece de forma transparente, mediante interfaces, la información generada por una aplicación de JDERobot para que podamos dar órdenes al robot. Estas órdenes se realizan mediante una serie de variables compartidas, las cuales se van modificando en función de si lo que queremos es conocer la información que nos proporciona un sensor simulado en Gazebo, o lo que queremos es queremos mandar órdenes a los actuadores del robot Nao simulado.

El *driver* Gazebo es capaz de comunicarse con el simulador Gazebo gracias a la libreria libgazebo, una libreria escrita en C que permite que otros programas se comuniquen e interaccionen con el simulador Gazebo.

El esquema de comunicación entre Gazebo y JDERobot, más concretamente entre el robot Nao simulado y las aplicaciones de JDERobot podemos observarlo en la figura 4.1(a)




(a)

***Figura 4.1:*** *Esquema(a).*

## 4.2. Soporte del Robot Nao en Gazebo

Como ya explicamos en el capítulo 3 un modelo de robot se compone de varios cuerpos, unidos entre sí mediante articulaciones. Los cuerpos se pueden crear mediante formas geométricas básicas como un hexaedro, una esfera, un cilindro, etc. A estas geometrías les puedes dar el tamaño que necesites para tu modelo, al igual que la masa. Estos cuerpos se unen entre sí mediante articulaciones, con las cuales, además, podemos darle una o varias direcciones de giro. Con ello se otorga a nuestro modelo simulado diferentes grados de libertad. Para poder usar el modelo que hemos creado es necesario registrar el *plugin* de nuestro modelo para poder cargarlo en nuestro mundo simulado. Esto se hace como vemos en la lista de código 4.1.

```
1
2 // Register the model
3 GZ_REGISTER_PLUGIN("Nao", Nao)
```

***Listing 4.1:*** *Registro del plugin del modelo Nao*

Con esto el modelo que creemos ya puede ser usado en cualquier mundo en Gazebo.

Una vez creada la parte mecánica del robot le damos una apariencia más cercana a la del robot Nao real. Para ello, iremos diseñando con Blender cada una de las distintas partes del cuerpo en 3D por separado. El hecho de construir el modelo en 3D por partes se debe a la forma en la que ha sido creado en Gazebo, pues por cada cuerpo en Gazebo se carga una de las pieles del robot. Para poder incorporarle una piel lo primero que debemos crear es la opcion de cargarla en el mundo, esto lo hacemos añadiendo a nuestro código en Gazebo las líneas que observamos en la lista 4.2, en nuestro ejemplo cargamos la piel de la cabeza. Lo primero que se hace es cargar el archivo *.3ds* y después aplicarselo al modelo.

```
1
2  // Apply a skin to body
3  skinFile = node->SearchFilename("skinFileHead", "./", NULL);
4
5  if (skinFile)
6  {
7    printf("loading skin file [%s]", skinFile);
8    if (geom->SetSkinFile(skinFile) != 0)
9    {
10     printf("unable to load skin file");
11     return -1;
12   }
13 }
```

***Listing 4.2:*** *Código carga de la piel en Gazebo*

Una vez hecho esto lo siguiente es cargar la piel cuando inicializamos el mundo, esto se hace en nuestro archivo *.world* añadiendo las siguientes líneas que vemos en la lista 4.3. Al igual que anteriormente el ejemplo es para la carga de la piel de la cabeza.

```
1
2    <skinFileHead>head_eyes.3ds</skinFileHead>
3    <skinScaleHead>0.125 0.125 0.125</skinScaleHead>
4    <skinXyzHead>0.005 0 0</skinXyzHead>-->
5    <skinRpyHead>0 0 90</skinRpyHead>
```

***Listing 4.3:*** *Código carga de la piel en Gazebo*

Para cargar la piel no es estrictamente necesario hacerlo a la hora de generar el Mundo, sino que se podría hacer en la creación del modelo en Gazebo. Sin embargo, decidimos hacerlo así para que la piel del robot sea independiente de su esqueleto, de este modo, basandonos en el mismo esqueleto de un robot Humanoide podriamos ponerle la apariencia del robot Nao azul, del rojo o de cualquier otro tipo de robot humanoide del que hayamos creado sus pieles correspondientes.

Para que el robot sea completamente funcional, hace falta añadirle los interfaces que nos permitan comunicar el simulador Gazebo con alguna aplicación exterior, en nuestro caso con una aplicación de JDERobot.

El primer interfaz que introducimos es el que nos permite la intercomunicación con las articulaciones del robot Nao. A lo largo del capítulo iremos viendo como se crean y se le añaden las articulaciones a nuestro modelo, ahora veremos como permitir recoger los datos que nos muestren y mandarles nosotros órdenes para que dichas articulaciones se muevan.

Se crea y se abre el interfaz para poder intercomunicarnos con dichas articulaciones, para ello necesitamos introducir las líneas de código que vemos en la lista 4.4

```
1
2  // Create and open a joint interface
3  this->joint_iface = gz_joint_alloc ();
4  if ( gz_joint_create (this->joint_iface, this->world->gz_server, this->GetId(),
5                        "Nao", this->GetIntId(), this->GetParentIntId()) != 0)
6  {
7    return -1;
8  }
```

***Listing 4.4:*** *Código creación y apertura del interfaz para las articulaciones*

Una vez creado y abierto debemos inicializar los datos de la interfaz como vemos en la lista 4.5 , de este modo le indicamos de que tipo es cada articulación(bisagra, Universal, etc), en que posición se encuentra inicialmente, que dirección o direcciones de giro tienen, etc.

Gracias a este interfaz y usando la libreria libgazebo podemos acceder a los datos de las articulaciones desde una aplicación externa al simulador Gazebo. Con esto ya tendríamos nuestro primer interfaz listo para comunicarse mediante libgazebo con aplicaciones externas al simulador, ahora pasamos a detallar cómo se crean las distintas partes del modelo en Gazebo y las articulaciones de que consta dicha interfaz.

```
1
2   // Set the number of joints in this model
3   this->joint_iface->data->joint_count = JOINT_COUNT;
4
5   // Fill in the joint information
6   for (int i=0; i<JOINT_COUNT; i++)
7   {
8     this->joint_iface->data->joints[i].type = this->joints[i]->GetType();
9
10    vec = this->joints[i]->GetAnchor();
11    this->joint_iface->data->joints[i].anchor[0] = vec.x;
12    this->joint_iface->data->joints[i].anchor[1] = vec.y;
13    this->joint_iface->data->joints[i].anchor[2] = vec.z;
14
15    if (this->joints[i]->GetType() == dJointTypeHinge)
16    {
17      hjoint = ((HingeJoint*)this->joints[i]);
18      vec = hjoint->GetAxis();
19      this->joint_iface->data->joints[i].axis[0] = vec.x;
20      this->joint_iface->data->joints[i].axis[1] = vec.y;
21      this->joint_iface->data->joints[i].axis[2] = vec.z;
22
23      this->joint_iface->data->joints[i].angular_velocity = hjoint->GetAngleRate();
24      this->joint_iface->data->joints[i].angle = hjoint->GetAngle();
25
26    }
27    else if (this->joints[i]->GetType() == dJointTypeUniversal)
28    {
29      ujoint = ((UniversalJoint*)this->joints[i]);
30      vec = ujoint->GetAxis1();
31      this->joint_iface->data->joints[i].axis[0] = vec.x;
32      this->joint_iface->data->joints[i].axis[1] = vec.y;
33      this->joint_iface->data->joints[i].axis[2] = vec.z;
34
35      vec = ujoint->GetAxis2();
36      this->joint_iface->data->joints[i].axis2[0] = vec.x;
37      this->joint_iface->data->joints[i].axis2[1] = vec.y;
38      this->joint_iface->data->joints[i].axis2[2] = vec.z;
39
40      this->joint_iface->data->joints[i].angular_velocity = ujoint->GetAngleRate1();
41      this->joint_iface->data->joints[i].angle = ujoint->GetAngle1();
42
43      this->joint_iface->data->joints[i].angular_velocity2 = ujoint->GetAngleRate2();
44      this->joint_iface->data->joints[i].angle2 = ujoint->GetAngle2();
45
46
47    }
```

***Listing 4.5:*** *Código inicialización del interfaz para las articulaciones*

### 4.2.1. Tronco

Para la creación de un cuerpo en Gazebo ya hemos ido comentado que es necesario construirloa base de figuras geometricas básicas como esferas, cilindros, hexaedros, etc. Con esta última comenzamos la construcción de nuestro modelo. Mediante un hexaedro, el cual tiene unas dimensiones de 100x196x50 mm.(Alto x Ancho x Fondo) y una masa de 1217,1 gramos construimos el tronco, nuestra base para la creación del robot Nao. A partir de este cuerpo iremos construyendo el resto de las partes que componen el robot Nao, como los brazos, las piernas y la cabeza. Podemos ver el código para su creacion en la lista 4.6.

```
1
2  // Create the torso
3    this->torso = new Body( this->world, "torso" );
4    this->torso->SetPosition(GzVectorSet(0.00, 0.00, torsoSize.z * 0.5));
5    this->torso->SetAngularVel(GzVectorSet(0,0,0));
6    this->torso->SetLinearVel(GzVectorSet(0,0,0));
7    this->AddBody( this->torso, false );
8
9    geom = new BoxGeom( this->torso, this->modelSpaceId, torsoSize.x, torsoSize.y,
10       torsoSize.z);
11   geom->SetRelativePosition(GzVectorSet(0, 0, torsoSize.z * 0.5) );
12   geom->SetMass( 1.2171 ); // 1217.1 g.
13   geom->SetColor( GzColor(0.8, 0.8, 0.8) );
```

***Listing 4.6:*** *Código de creación de la parte del tronco*

El hexaedro que hemos construido para simular el tronco del robot Nao no se parece en nada al tronco del robot real, por lo que nos hace falta crearle una piel que tenga una apariencia más cercana a la real.

Para diseñar el tronco en Blender también comenzamos con un hexaedro rectangular, el cual vamos estirando y redondeando a nuestro gusto para que se vaya pareciendo lo más posible a la forma que tiene el cuerpo del robot Nao real. Con Blender podemos incluso unir varios modelos, que fue lo que hicimos en la creacion del cuerpo de nuestro robot simulado. Creamos la parte del pecho por un lado mediante un hexaedro, las hombreras por otro usando para crearlas una esfera que posteriormente dividimos por la mitad, la parte de la cadera por otro, la cual la unimos a la parte del pecho mediante la creacion de un cilindro que también moldeamos nostros mismos. El color principal de estas figuras, así como de todas las creadas en Blender para nuestro robot Nao simulado, es el blanco, pero cada una de ellas, además, consta de detalles en color Azul. Como podemos ver en la figura (4.2(a)) el robot Nao posee las hombreras de color y, además, una banda en el pecho, también del mismo color que las hombreras, donde el robot real incorpora los sensores de ultrasonido.

Un detalle importante a destacar es que en la creación del modelo en 3D con Blender no hace falta ceñirnos a los tamaños reales del robot. Nosotros podemos

(a)

***Figura 4.2:*** *Cuerpo robot Nao creado con Blender(a).*

crear un modelo con las dimensiones que queramos, generamos el archivo *.3ds* que nos permite cargar Gazebo y es dentro del código de Gazebo, o en el archivo que carga nuestro Mundo cuando le decimos que escala debe coger para que se adapte a nuestro modelo de robot humanoide simulado.

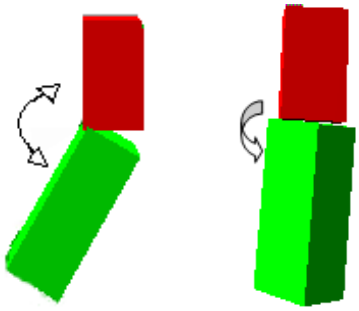
En el caso del cuerpo, nuestro robot Nao simulado no incorpora ningun interfaz software.

### 4.2.2. Brazos

El brazo de nuestro robot se compone de la parte del húmero, la parte del cúbito y radio y la mano. La parte del húmero la hemos creado construyendo un hexaedro rectangular con unas dimesiones de 90x50x50 mm y con una masa de 163 gramos, la parte del cúbito y radio tambien la creamos construyendo un hexaedro rectangular, pero esta vez con unas dimensiones de 135x50x50 mm. y una masa de 87 gramos. Estos 2 cuerpos se unen mediante la articulación del codo, esta articulación tiene una peculiaridad y es que a parte de que nos da el grado de libertad normal del codo, el cual nos permite doblar y estirar el brazo(figura 4.3(a)), también dispone de otro grado de libertad más que nos permite girar el antebrazo con respecto del húmero (figura 4.3(b)).

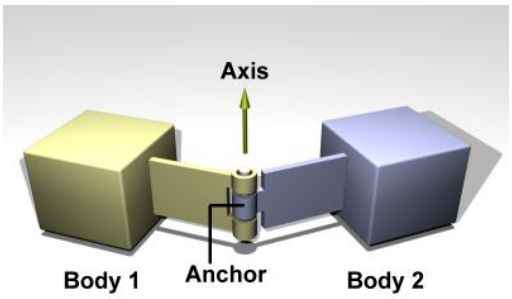
Para conseguir estos 2 grados de libertad del codo debemos utilizar dos articulaciones de tipo bisagra (figura 4.4(a)), pues es la única forma de conseguir que al girar el antebrazo gire con él la otra articulación del codo que nos permite estirar y doblar el brazo. Gazebo no nos permite unir dos cuerpos con más de una articulación al mismo tiempo, con lo que debemos introducir entre nuestro brazo y nuestro antebrazo otro cuerpo. Este nuevo cuerpo creado para poder recrear los grados de libertad del codo que posee el robot Nao real, tiene un tamaño lo suficientemente pequeño para que no se aprecie de forma visual y una masa también lo suficientemente pequeña para que sea despreciable en el computo global de la masa de nuestro robot simulado.

Así, nuestro brazo estaría unido a este cuerpo mediante una articulación de tipo bisagra que simularía una dirección *Yaw* con una amplitud de movimiento de -120



(a) (b)

***Figura 4.3:*** *Movimiento Yaw del codo(a). Movimiento Roll del codo (b)*




(a)

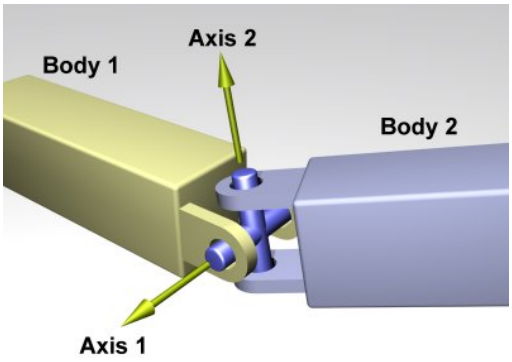
***Figura 4.4:*** *Articulación de tipo bisagra(a).*

hasta 120 grados. Por el otro lado, el cuerpo intermedio estaría unido mediante otra articulación de tipo bisagra con nuestro antebrazo, en este caso tendríamos una amplitud de -90 a 0 grados para un movimiento *Roll*.

Los brazos estan unidos al tronco, en este caso volvemos a tener otros dos grados de libertad por cada hombro. Esta vez, las direcciones de estos dos grados de libertad son: *Pitch y roll*.

El movimiento *Pitch* del hombro es capaz de recorrer desde -120 grados hasta 120 grados y el movimiento *roll*, el cual separa el brazo del tronco, desde 0 hasta 95 grados.

Para conseguir estos 2 grados de libertad en nuestro robot Nao simulado utilizamos una articulación tipo Universal (figura 4.5(a)). En la figura del ejemplo el *Body1* sería nuestro tronco y el *Body2* nuestro brazo. Para crear la articulación lo primero que debemos indicar es de qué tipo es dicha articulación, lo siguiente sería indicarle a Gazebo cuales son los dos cuerpos a unir por dicha articulación, la dirección del movimiento (*Pitch, Yaw, Roll*) y por último los parámetros concernientes a ese tipo de movimiento como la amplitud del movimiento, es decir, el grado mínimo y máximo que puede alcanzar. Podemos ver un ejemplo en la lista 4.7




(a)

***Figura 4.5:*** *Articulación de tipo Universal(a).*

```
1
2   // Left shoulder joint
3   this->joints[L_SHOULDER] = new UniversalJoint(this->world);
4   this->joints[L_SHOULDER]->Attach(this->torso, this->leftUpperArm);
5   anchor = this->leftUpperArm->GetCoMPose().pos;
6   this->joints[L_SHOULDER]->SetAnchor(anchor);
7   ((UniversalJoint*)this->joints[L_SHOULDER])->SetAxis1(GzVectorSet(0,1,0));
8   ((UniversalJoint*)this->joints[L_SHOULDER])->SetAxis2(GzVectorSet(0,0,1));
9
10  this->joints[L_SHOULDER]->SetParam( dParamBounce, 0);
11  this->joints[L_SHOULDER]->SetParam( dParamBounce2, 0);
12  this->joints[L_SHOULDER]->SetParam( dParamLoStop, DTOR(-120));
13  this->joints[L_SHOULDER]->SetParam( dParamLoStop2, DTOR(0));
14  this->joints[L_SHOULDER]->SetParam( dParamHiStop, DTOR(120.0) );
15  this->joints[L_SHOULDER]->SetParam( dParamHiStop2, DTOR(95.0) );
16  this->joints[L_SHOULDER]->SetParam( dParamFMax, 1.1 );
17  this->joints[L_SHOULDER]->SetParam( dParamFMax2, 1.1 );
```

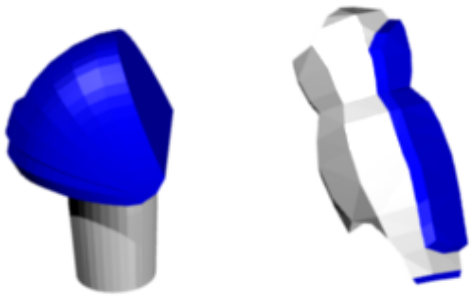
***Listing 4.7:*** *Código de creación de articulación del hombro*

Así, ya tenemos el brazo construido en Gazebo mediante figuras geométricas básicas(figura 4.7(a)) y unido al tronco, con lo que el siguiente paso es diseñar en Blender la piel del brazo, de forma que sea visualmente lo más parecido al robot Nao real.

Igual que en la construcción en Gazebo, nuestro brazo consta de dos partes, cada una de ellas se comienza a construir mediante la misma forma geométrica que hemos usado en Gazebo, en este caso 2 hexaedros rectangulares, lo cuales vamos moldeando a nuestro gusto para conseguir la deseada apariencia del robot Nao real. En el caso de la parte superior del brazo, además, hay que colorear de azul las hombreras de que consta, las cuales se crearon de forma independiente y fueron unidas posteriormente. Estas hombreras esta moldeadas a partir de una esfera, la cual dividimos para obtener la forma que se observa en la figura (4.6(a)).

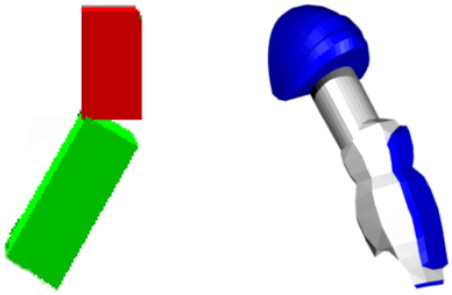
El antebrazo también tiene alguna parte coloreada de azul cerca de la parte de la

muñeca como podemos observar en la figura (4.6(b)). Podemos observar claramente la diferencia entre el brazo completo sin pieles (figura 4.7(a)) y con pieles (figura 4.7(b))



(a) (b)

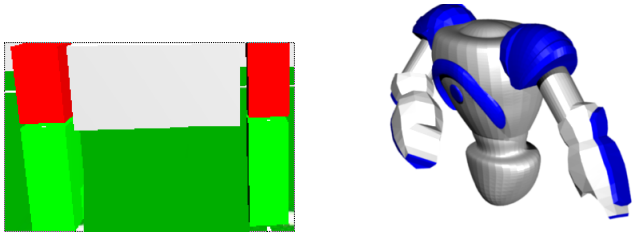
***Figura 4.6:*** *Diseño 3D del hombro(a). Diseño 3D del antebrazo (b).*



(a) (b)

***Figura 4.7:*** *Comparación Brazo: Sin pieles(a). Con pieles (b).*

Exceptuando la cabeza ya tenemos construido el esqueleto de nuestro robot simulado de cintura para arriba, construido mediante figuras geométricas básicas como observamos en la figura 4.8(a) y además con la piel creada en Blender para darle una apariencia visual más cercana al robot Nao real 4.8(b).



(a) (b)

***Figura 4.8:*** *Tronco y extremidades superiores: Sin pieles(a). Con pieles(b)*

Los brazos disponen en total de de 4 articulaciones: hombro izquierdo, hombro derecho, codo izquierdo y codo derecho, lo que dota al robot de 8 grados de libertad

en los brazos. Las interfaces con las son que podremos controlar estas articulaciones vienen definidas por las variables: (*L_SHOULDER*), (*R_SHOULDER*),(*R_ELBOW_YAW y R_ELBOW*) y (*L_ELBOW_YAW y L_ELBOW*). Gracias a ellas podremos devolver los datos de las articulaciones cuando nos los pidan desde una aplicación externa. Para ello tenemos que recorrernos todas las articulaciones para ir recogiendo dichos datos teniendo en cuenta el tipo de articulación en cada caso para saber si debemos devolver uno o dos parámatros. Para devolver los datos de las articulaciones, son necesarias las líneas de código de la lista 4.8

```
1
2  // Fill in the joint information
3  for (int i=0; i<JOINT_COUNT; i++)
4  {
5    if (this->joints[i]->GetType() == dJointTypeHinge)
6    {
7      this->joint_iface->data->joints[i].angular_velocity =
8                    ((HingeJoint*)this->joints[i])->GetAngleRate();
9      this->joint_iface->data->joints[i].angle =
10                    ((HingeJoint*)this->joints[i])->GetAngle();
11   }
12
13   else if (this->joints[i]->GetType() == dJointTypeUniversal)
14   {
15     this->joint_iface->data->joints[i].angular_velocity =
16                   ((UniversalJoint*)this->joints[i])->GetAngleRate1();
17     this->joint_iface->data->joints[i].angular_velocity2 =
18                   ((UniversalJoint*)this->joints[i])->GetAngleRate2();
19
20     this->joint_iface->data->joints[i].angle =
21                   ((UniversalJoint*)this->joints[i])->GetAngle1();
22     this->joint_iface->data->joints[i].angle2 =
23                   ((UniversalJoint*)this->joints[i])->GetAngle2();
24   }
25
26 }
```

***Listing 4.8:*** *Código para devolver los datos de las articulaciones*

Además, también con este interfaz podemos dar valores desde una aplicación externa y que las articulaciones se muevan hasta un determinado ángulo. En la lista 4.9 podemos observar las líneas de código que hacen falta para realizar esta tarea. Hay que tener en cuenta que se recorren todas las articulaciones, pero sólo se moverán aquellas en las que el ángulo de destino sea distinto al ángulo de partida, esto lo comprobamos con el subprograma *SetJointVelocity* en dónde le damos velocidad, o mas bien activamos, las articulaciones que deben moverse (lista 4.10).

Así, quedarían definidas las interfaces de las articulaciones del brazo, con las que podríamos recrear los 8 grados de libertad que posee el robot Nao en los brazos.

```
1
2
3   // Process all joints
4   for (int i=0; i<JOINT_COUNT; i++)
5   {
6     if (this->joints[i]->GetType() == dJointTypeHinge)
7     {
8       angle = ((HingeJoint*)this->joints[i])->GetAngle();
9       cmd_angle = this->joint_iface->data->joints[i].cmd_angle;
10
11      this->SetJointVelocity(this->joints[i], dParamVel, angle, cmd_angle);
12    }
13
14    else if (this->joints[i]->GetType() == dJointTypeUniversal)
15    {
16
17      angle = ((UniversalJoint*)this->joints[i])->GetAngle1();
18      cmd_angle = this->joint_iface->data->joints[i].cmd_angle;
19
20
21      this->SetJointVelocity(this->joints[i], dParamVel, angle, cmd_angle);
22
23      angle = ((UniversalJoint*)this->joints[i])->GetAngle2();
24      cmd_angle = this->joint_iface->data->joints[i].cmd_angle2;
25
26      this->SetJointVelocity(this->joints[i], dParamVel2, angle, cmd_angle);
27    }
28
29
30  }
```

***Listing 4.9:*** *Código para actualizar las articulaciones con datos recibidos de aplicación externa*

```
1
2  double velocity = 0.0;
3
4  // Move the joint if the commanded angle doesn't match the current angle
5  if (fabs(angle - cmd_angle) > 1e-2)
6  {
7    if (angle < cmd_angle)
8      velocity = 1.0;
9    else
10     velocity = -1.0;
11 }
12
13 if (joint->GetType() == dJointTypeHinge)
14   ((HingeJoint*)joint)->SetParam(param, velocity);
15 else
16   ((UniversalJoint*)joint)->SetParam(param, velocity);
```

***Listing 4.10:*** *Código para activar las articulaciones que deben realizar movimiento*

### 4.2.3. Piernas

Ahora llega la parte más compleja, que es poner el tronco sobre dos soportes(las dos piernas de nuestro robot) y que éste se mantenga en equilibrio.
Para ello el tronco lo unimos a la cadera mediante una articulacion de tipo bisagra, con esta articulacion realizamos un movimiento combinado *Yaw-Pitch*. Esta articulación mueve en ese sentido las 2 piernas a la vez, igual que lo hace el robot Nao real. Además de este grado de libertad, el fémur se une a la cadera con otra articulación más, esta vez de tipo Universal. Con esta articulación conseguimos que el robot tenga otros 2 grados de libertad adicionales, uno de ellos con una dirección *Pitch* y una amplitud de -100 a 25 grados y el otro con una dirección *Roll* y una amplitud desde -25 hasta 45 grados.

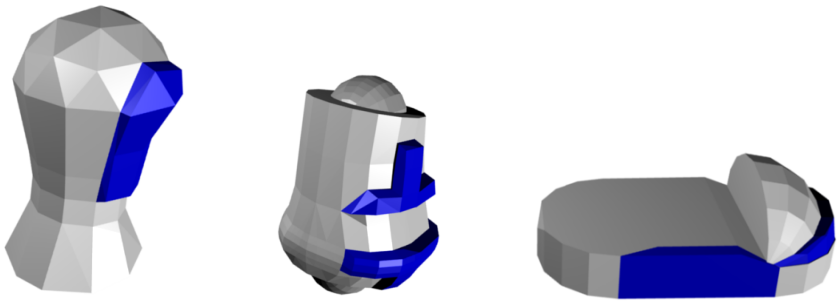
Así ya estaría unido nuestro fémur, que mide 100x65x65 mm. y pesa 533 gramos, con el tronco. Ahora tendriamos que unir esta parte con la de la tibia-peroné, la cual tiene también unas dimensiones de 100x65x65 mm. y una masa de 423 gramos.

Para unir estas dos partes y así poder dotar al robot de otro grado de libertad más con una amplitud de 0 a 130 grados en dirección *Pitch* utilizamos, una vez más, una articulación de tipo bisagra para conseguir recrear la articulación de la rodilla.

Una vez unidas estas dos partes por medio de la articulación de tipo bisagra lo siguiente es unir toda la pierna con el pie y añadirle la articulación del tobillo. El pie tiene una masa de 158 gramos y unas dimensiones de 46x102x160 mm. El tipo de articulación utilizado para esta unión es de tipo universal, ya que en la articulación del tobillo el robot consta de dos grados de libertad. Uno de ellos con dirección *Pitch* y el otro con dirección *Roll*.

El siguiente paso, al igual que venimos haciendo con las anteriores partes del robot simulado es diseñar y modelar en Blender esta última parte del robot creada. Esta vez tenemos que crear tres modelos, el muslo, la tibia y el pie. En la construcción

del muslo partimos nuevamente de un hexaedro, el cual moldeamos y deformamos hasta que toma la forma adecuada (figura 4.9(a)). Del mismo modo construimos



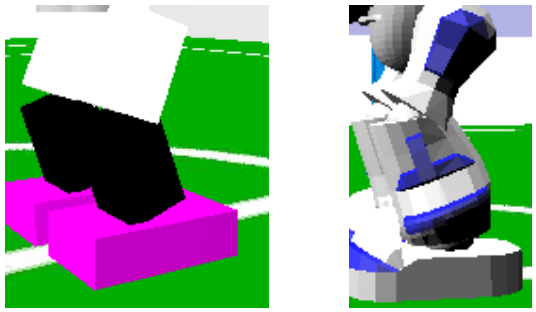
(a) (b) (c)

***Figura 4.9:*** *Partes de la pierna creadas con Blender: Muslo(a). Tibia (b). Pie (c)*

la parte de la tibia, cogemos un hexaedro y lo moldeamos. En este caso la tibia, además, consta de un par de figuras más en el lateral que posteriormente pintaremos de azul(figura 4.9(b)).

En cuanto al pie tenemos dos partes, lo que sería la suela, que está construida a partir de un hexaedro y la puntera del pie, que tiene color azul y la cual se construyó a partir de una esfera y fue posteriormente unida a la suela (figura 4.9(c)).

En las figuras 4.10(a) y 4.10(b) podemos observar la diferencia entre el modelo creado con figuras geometricas básicas sin añadirle las pieles, y una vez añadidas.



(a) (b)

***Figura 4.10:*** *Piernas: Sin piel(a). Con piel (b)*

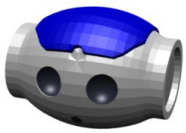
Las interfaces de esta parte del modelo vienen determinadas por las variables: *WAIST*, *L_HIP*, *L_KNEE*, *L_ANKLE*, *R_HIP*, *R_KNEE* y *R_ANKLE*. A estas interfaces se accede de igual forma a como se acceden a las que corresponden con las de los brazos. En este caso, el robot dispone en las piernas de 10 grados de libertad.

Ya tendríamos nuestro robot simulado con tronco, brazos, piernas, y todas las articulaciones disponibles.

### 4.2.4. Cabeza

Para completar nuestro robot humanoide faltaría por añadir la cabeza para, de este modo, tener el esqueleto del robot simulado al completo. La cabeza pasamos a construirla mediante una esfera de 50mm de diámetro con una masa de 401 gramos.Va unida al tronco mediante una articulación de tipo Universal, así podemos mover la cabeza en las direcciones *pan y tilt*. Mediante esta articulación podemos girar la cabeza de derecha a izquierda *pan* desde -120 hasta 120 grados, y de ariba a abajo *tilt* desde -45 grados hasta 45 grados.

Para conseguir uno de nuestros objetivos que era que se pareciese lo más posible al robot Nao real en apariencia, modelamos en 3D la cabeza(figura 4.11(a)) . La cabeza fue creada basándonos en un cilindro al cual le fuimos estirando algunas partes, añadiendo o quitando otras mediante uniones e intersecciones, como por ejemplo en la parte de los ojos, que son dos esferas, las cuales añadimos a nuestro diseño en 3D mediante una operación booleana de diferencia entre el cilindro y las esferas. Mediante la operacion booleana de unión se le añadió también la parte en color azul que el robot Nao real lleva en la cabeza.



(a)

***Figura 4.11:*** *Cabeza creada con Blender (a)*

Ahora sí que tenemos el esqueleto del robot Humanoide Nao completamente simulado. Por un lado tenemos el esqueleto con figuras geométricas básicas en Gazebo, y por otro tenemos los archivos de las partes diseñadas en 3D creadas con Blender que nos permitirán darle al robot Nao una apariencia más cercana al robot Nao real.

El robot Nao real, además, lleva incorporada en la cabeza una cámara, la cual es utilizada en la Robocup para autolocalizarse, localizar a los compañeros, etc. Nuestro robot Nao simulado también va a constar de esta cámara, asi que debemos añadir el sensor a nuestro modelo Nao simulado. Este sensor lo hemos creado basándonos en la cámara SonyVID30, sensor que ya está creado en Gazebo y el cual, por ejemplo usa el modelo Pioneer2DX. Esta cámara por defecto nos proporciona imágenes de 320x240 píxeles, con un campo de visión horizontal($HFOV$) de 60 grados y un refresco de 10 frames por segundo.

Todos estos valores por defecto pueden ser modificados al cargar nuestro mundo, al igual que otros parámetros de configuración como los planos de corte, es decir, a que distancia mínima y máxima la cámara es capaz de capturar imágenes. Otra opción que nos permite configurar es si queremos guardar los fotogramas en una determinada ruta que le proporcionemos, aunque por defecto esta opción esta deshabilitada.

Una vez que tenemos definidos todos estos parámetros como vemos en la lista 4.11 pasamos a incorporarla en la cabeza de nuestro robot, poniendola en las coordenadas pertinentes. Para incorporarla basta con incluir las líneas de código que se pueden ver en la lista 4.12.

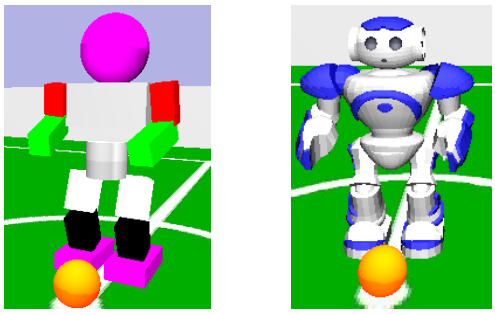
Definidos estos parámetros ya tenemos el sensor incorporado en nuestra cabeza de forma totalmente transparente, es decir, que de la cámara SonyVID30 tan solo nos hemos quedado, por decirlo de algun modo, con la lente. No nos interesaba añadir la cámara tal cual a nuestro modelo, con carcasa incluida, pues sería añadirle a nuestro robot simulado un cuerpo con una masa que no tiene el robot Nao real.

Hecho esto, es hora de crear el interfaz que nos permita acceder a los datos de la cámara, y poder ver así lo que nuestro robot Nao en Gazebo esta observando. Al igual que hemos hecho con el interfaz de las articulaciones lo primero es crear y abrir dicha interfaz (lista 4.13). Lo siguiente es, al igual que con el interfaz anterior, inicializar los parámetros con los que ha sido creada la cámara (lista 4.14).

En el caso de la interfaz de la cámara tan solo puede mandar datos a las aplicaciones externas que los requieran. Para que, mediante libgazebo, podamos acceder a los datos que nos devuelve la cámara debemos introducir las líneas de código observables en la lista 4.15

Así, ya tendriamos el sensor de la cámara incorporado en nnuestro robot Nao simulado con su interfaz de comunicación incluida, por lo que tan solo nos quedaría añadirle las interfaces para controlar los 2 grados de libertad que el robot posee en el cuello. Estas dos interfaces (controladas por la variable *HEAD*) se añadirían de forma similar a como hemos hecho con las interfaces de los brazos y de las piernas.

Ahora sí que tenemos el robot Humanoide Nao completamente diseñado y simulado en Gazebo (figura 4.12(a)), con la apariencia igual a la del robot Nao real (figura 4.12(b)). Tenemos por tanto el esqueleto completo del robot junto con los actuadores de las articulaciones y uno de los sensores más importantes de que dispone: la cámara.



(a) (b)

***Figura 4.12:*** *Modelo completo: Sin piel(a). Con piel (b)*

```
1
2  // Get the time between camera updates
3    this->cameraUpdatePeriod = 1.0 / (node->GetDouble("updateRate", 10) + 1e-6);
4    this->cameraUpdateTime = -this->cameraUpdatePeriod;
5
6    // Lens field of view
7    this->hfov = node->GetAngle("hfov", DTOR(60));
8
9    // Zoom limits
10   this->zoomMin = node->GetTupleAngle("zoomLimits", 0, 1);
11   this->zoomMax = node->GetTupleAngle("zoomLimits", 1, 10);
12
13   // Gain term on PTZ controller
14   this->motionGain = node->GetDouble( "motionGain", 2.0 );
15   this->zoomGain = node->GetDouble( "zoomGain", 2.0 );
16
17   // Initial camera settings
18   this->cmdZoom = this->zoomMin;
19   this->zoom = this->cmdZoom;
20
21   // Get image dimensions
22   imgWidth = node->GetTupleInt("imageSize", 0, 320);
23   imgHeight = node->GetTupleInt("imageSize", 1, 240);
24
25   // Get camera parameters
26   nearClip = node->GetLength("nearClip", 0.050, 0);
27   farClip  = node->GetLength("farClip", 50.0, 0);
28   method = node->GetString("renderMethod", "auto");
29   zDepth = node->GetInt("zBufferDepth", 16);
30   if (((zDepth % 8) != 0) || (zDepth < 8) || (zDepth > 32))
31     zDepth = 16;
32
33   // Initialize the sensor
34   if (this->camera->Init(imgWidth, imgHeight, this->hfov * this->cmdZoom,
35                          nearClip, farClip, method, zDepth) != 0)
36     return -1;
37
38   // Path for saving frames
39   this->camera->SetSavePath(node->GetString("savePath", "."));
40   this->camera->EnableSaveFrame(node->GetString("saveFrames", false));
```

***Listing 4.11:*** *Código parámetros de la cámara*

```
1
2 pos = this->head->GetPosition();
3     pos = GzCoordPositionAdd(GzVectorSet(0.10, 0, 0), pos, rot);
4
5     this->camera->SetFOV(this->hfov / this->zoom);
6     this->camera->SetPose(GzPoseSet(pos, rot));
```

***Listing 4.12:*** *Código colocacion de la cámara en la cabeza*

```
1
2   // Camera interface
3   assert(this->cameraIface);
4   if (gz_camera_create(this->cameraIface, this->world->gz_server, this->GetId(),
5                        "NaoCamera1", this->GetIntId(), this->GetParentIntId()) !=0)
6     return -1;
```

***Listing 4.13:*** *Código creación y apertura del interfaz para la cámara*

```
1
2   // Set image dimensions (needed by GUI)
3   gz_camera_lock(this->cameraIface, 1);
4   data = this->cameraIface->data;
5   this->camera->GetImageSize(&width, &height);
6   data->width = width;
7   data->height = height;
8   gz_camera_unlock(this->cameraIface);
9
10  // Set initial rendering options
11  renderOpts = new RenderOptions();
12  this->camera->GetRenderOptions(renderOpts);
13  renderOpts->displaySkins = true;
14  this->camera->SetRenderOptions(renderOpts);
```

***Listing 4.14:*** *Código inicialización del interfaz de la cámara*

```
1
2  gz_camera_lock(this->cameraIface, 1);
3
4  data = this->cameraIface->data;
5
6  // Get image dimensions
7  this->camera->GetImageSize(&width, &height);
8
9  data->time = this->world->GetSimTime();
10 data->width = width;
11 data->height = height;
12 data->image_size = width * height * 3;
13
14 // Make sure there is room to store the image
15 assert(data->image_size <= sizeof(data->image));
16
17 // Copy the pixel data to the interface, but flip the y axis
18 len = width * 3;
19 src = this->camera->GetImageData() + (height - 1) * len;
20 dst = data->image;
21 for (i = 0; i < height; i++, src -= len, dst += len)
22   memcpy(dst, src, len);
```

***Listing 4.15:*** *Código para devolver los datos de las articulaciones*

El siguiente paso, que detallamos en el siguiente punto de la memoria, es dar soporte a este robot que hemos diseñado para que podamos darle movimientos, capturar datos del sensor que incorpora, etc. desde una aplicación de JDERobot.

## 4.3. Driver Gazebo

El driver Gazebo para JDERobot nos permite la interconexión entre una aplicación de JDERobot y el simulador Gazebo. Para ello, necesitamos establecer una comunicación entre aplicación-*driver* y otra entre *driver*-simulador Gazebo. Este *driver* ejecuta un *thread* de forma iterativa, en el cual se encarga de recorrer todas las interfaces Gazebo preguntando en qué posición se encuentran las articulaciones en el simulador(lista 4.16), actualizando su copia local de las variables sensoriales para que puedan ser leídas por la aplicación. En ese mismo *thread*, además, va mandando órdenes a los actuadores de las articulaciones del modelo simulado a través de libgazebo, dándole los datos que la aplicación ha escrito en las variables de actuación (lista 4.17).

- **Comunicación aplicación-*driver***

  La comunicación entre una aplicación y el *driver* Gazebo se realiza mediante una serie de variables compartidas. Lo primero que debemos hacer es configurar el archivo de configuración de JDERobot para indicar qué interfaces del driver son las que vamos a utilizar para comunicar lsa aplicaciones de JDERobot

```
1
2     right_hip . enc_roll =((joint->data->joints[10].angle) /3.1416)*180;
3     right_hip . enc_pitch=(-(joint->data->joints[10].angle2) /3.1416)*180;
4     right_knee . enc_pitch=(-(joint->data->joints[11].angle) /3.1416)*180;
5     right_ankle . enc_pitch=(-(joint->data->joints[12].angle) /3.1416)*180;
6     right_ankle . enc_roll =((joint->data->joints[12].angle2) /3.1416)*180;
```

***Listing 4.16:*** *Código posición de las articulaciones de la pierna derecha*

```
1
2     joint->data->joints[10].cmd_angle =(right_hip.roll*3.1416)/180.0;
3     joint->data->joints[10].cmd_angle2 =(-right_hip.pitch*3.1416)/180.0;
4     joint->data->joints[11].cmd_angle =(-right_knee.pitch*3.1416)/180.0;
5     joint->data->joints[12].cmd_angle =(-(right_ankle.pitch)*3.1416)/180.0;
6     joint->data->joints[12].cmd_angle2 =(right_ankle.roll*3.1416)/180.0;
```

***Listing 4.17:*** *Código mandamos posición a las articulaciones de la pierna derecha*

con el robot Nao simulado en Gazebo (4.18). En nuestro caso las interfaces que vamos a necesitar son VarcolorA, para los datos de la cámara de nuestro robot, de la que no tuvimos que modificar el código pues VarcolorA ya estaba soportado por el driver Gazebo para obtención de imágenes de una cámara simulada en Gazebo.

El resto de interfaces que necesitamos son *left_arm*, *right_arm*, *left_leg* y *right_leg* para las articulaciones. El hecho de dividir en cuatro partes la interfaz de las articulaciones viene dado por, de alguna forma, seguir el mismo convenio que el utilizado en el *driver* naobody realizado por Francisco Miguel Rivas. De este modo, las aplicaciones que usen estas interfaces junto con el *driver* Naobody para comunicarse con el robot Nao real, pueden usar las mismas interfaces y el *driver* Gazebo para comunicarse con el robot Nao simulado sin tener que modificar el código de la aplicación sino tan solo el fichero de configuración de JDERobot. Consiguiendo así una alta compatibilidad con las aplicaciones ya creadas para JDERobot. Mediante las interfaces anteriormente

```
1
2  driver gazebo
3  provides varcolorA robot1
4  provides ptmotors
5  provides ptencoders
6  provides left_arm
7  provides right_arm
8  provides  left_leg
9  provides  right_leg
10 provides movement
11 end_driver
```

***Listing 4.18:*** *Código archivo configuración JDERobot*

mencionadas realizamos la intercomunicación entre el *driver* Gazebo y las aplicaciones de JDERobot, para esta intercomunicación exportamos algunas variables, las cuales nos sirven para leer datos y, además, para escribir datos que posteriormente serán pasados a las interfaces del modelo creado en Gazebo. La exportación de las variables se hace con el código que observamos en la lista 4.19.

```
1
2   if (serve_right_arm==1){
3       all[num_schemas].id = (int *) &right_arm_schema_id;
4       strcpy( all[num_schemas].name,"right_arm");
5       all[num_schemas].run = (runFn) right_arm_run;
6       all[num_schemas].stop = (stopFn) right_arm_stop;
7       printf(" %s schema loaded (id  %d)\n",all[num_schemas].name,num_schemas);
8       (*( all[num_schemas].id)) = num_schemas;
9       all[num_schemas].fps = 0.;
10      all[num_schemas].k =0;
11      all[num_schemas].state=slept;
12      all[num_schemas].terminate = NULL;
13      all[num_schemas].handle = NULL;
14      num_schemas++;
15
16      right_shoulder.clock=0;
17      right_elbow.clock=0;
18      myexport((char*)"right_arm",(char*)"right_shoulder",&right_shoulder);
19      myexport((char*)"right_arm",(char*)"right_elbow",&right_elbow);
20      myexport((char*)"right_arm",(char*)"id",&right_arm_schema_id);
21      myexport((char*)"right_arm",(char*)"run",(void *)right_arm_run);
22      myexport((char*)"right_arm",(char*)"stop",(void *)right_arm_stop);
23      myexport((char*)"right_arm",(char*)"move_speed",&move_speed);
24  }
```

***Listing 4.19:*** *Código exportación de variables*

En este punto ya tenemos todo listo para leer los datos que nos proporcionen las interfaces y para enviar datos mediante algunas de las variables exportadas, sin embargo, falta que las interfaces del simulador Gazebo, más concretamente las interfaces de nuestro modelo Nao simulado también se comuniquen con el *driver* Gazebo de JDERobot, bien para mandar los datos que la aplicación quiere leer o bien para recibir las órdenes que esta misma aplicación quiere mandar a los actuadores.

- **Comunicación *driver*-Simulador**

  La comunicación entre el *driver* Gazebo y el simulador se realiza gracias a la libreria libgazebo. Esta librería nos permite la comunicación entre las interfaces de nuestro modelo en Gazebo y el *driver*. Para comunicar las interfaces primero debemos establecer comunicación entre el *driver* y el simulador (lista 4.20).

```
1
2   /* Connection with Gazebo simulator */
3   printf ("Connecting to Gazebo Simulator\n");
4   gz_error_init (1, 9);
5
6   client = gz_client_alloc ();
7   if (gz_client_connect_wait (client, server_id, client_id) != 0)
8     {
9       fprintf (stderr, "gazebo: Error connecting to gazebo server\n");
10      exit (-1);
11    }
```

***Listing 4.20:*** *Código conexión entre el driver y el simulador*

Una vez realizada la conexión entre el simulador Gazebo y el *driver* Gazebo de JDERobot, miramos qué interfaces de comunicación entre el *driver* y la aplicación tienen que comunicarse con alguna interfaz en concreto del simulador.

En nuestro caso las interfaces *right_arm*, *left_arm*, *right_leg* y *left_leg* se comunican todas con la interfaz de las articulaciones de nuestro modelo en el simulador Gazebo. Cada una de ellas se comunica con una parte distinta de esta interfaz, o más bien con una serie de datos de la misma. Como su nombre indica la interfaz left_arm se comunicará con las articulaciones del brazo izquierdo, right_arm hará lo propio con las articulaciones del brazo derecho, left_leg establecerá comunicación con las articulaciones de la pierna izquierda y por último, right_leg será la encargada de comunicarse con las articulaciones de la pierna derecha. Todas estas comunicaciones hay que abrirlas introduciendo el código de la lista 4.21.

```
1
2   int right_arm_run(int father, int *brothers, arbitration fn){
3     if((serve_right_arm==1)&&(right_arm_active==0)){
4         printf(" shoulder_lrighteft  schema run (gazebo driver)\n");
5         all[right_arm_schema_id].father = father;
6         all[right_arm_schema_id].fps = 0.0;
7         all[right_arm_schema_id].k =0;
8        put_state(right_arm_schema_id,winner);
9        right_arm_active=1;
10        should_driver_restart ();
11    }
12    return 0;
13  }
```

***Listing 4.21:*** *Código comunicación de las interfaces*

Podemos ver la relación de interfaces entre el robot NAo simulado en Gazebo y las aplicaciones de JDERobot en la siguiente tabla 4.13(a)).

| RELACIÓN INTERFACES | |
|---|---|
| GAZEBO | JDEROBOT |
| R_Shoulder | Right_arm |
| R_Elbow | Right_arm |
| R_Elbow_Yaw | Right_arm |
| L_Shoulder | Left_arm |
| L_Elbow | Left _arm |
| L_Elbow_Yaw | Left _arm |
| WAIST | Left_leg & Right_leg |
| R_Hip | Right_leg |
| R_Knee | Right_leg |
| R_Ankle | Right_leg |
| L_Hip | Left _leg |
| L_Knee | Left _leg |
| L_Ankle | Left _leg |
| Camera | VarcolorA |
| HEAD | ptmotors & ptencoders |

(a)

***Figura 4.13:*** *Relación entre las interfaces de Gazebo y JDERobot (a).*

De este modo conseguimos dar soporte a las aplicaciones de JDERobot para que puedan manejar y controlar cualquier movimiento que realice el robot Humaoide Nao simulado en Gazebo.

# Capítulo 5

# Aplicaciones

Una vez descrito el diseño y la implementación, se expondrán en este capítulo una serie de aplicaciones creadas especialmente para validar la solución desarrollada y demostrar sus posibilidades. Se pueden ver varios videos de estas aplicaciones en el mediawiki[1] creado a lo largo del desarrollo de este proyecto.También permiten analizar con cierto detalle el correcto funcionamiento de nuestro robot Nao simulado y sus diferentes componentes.

## 5.1. Comportamiento mirar pelota

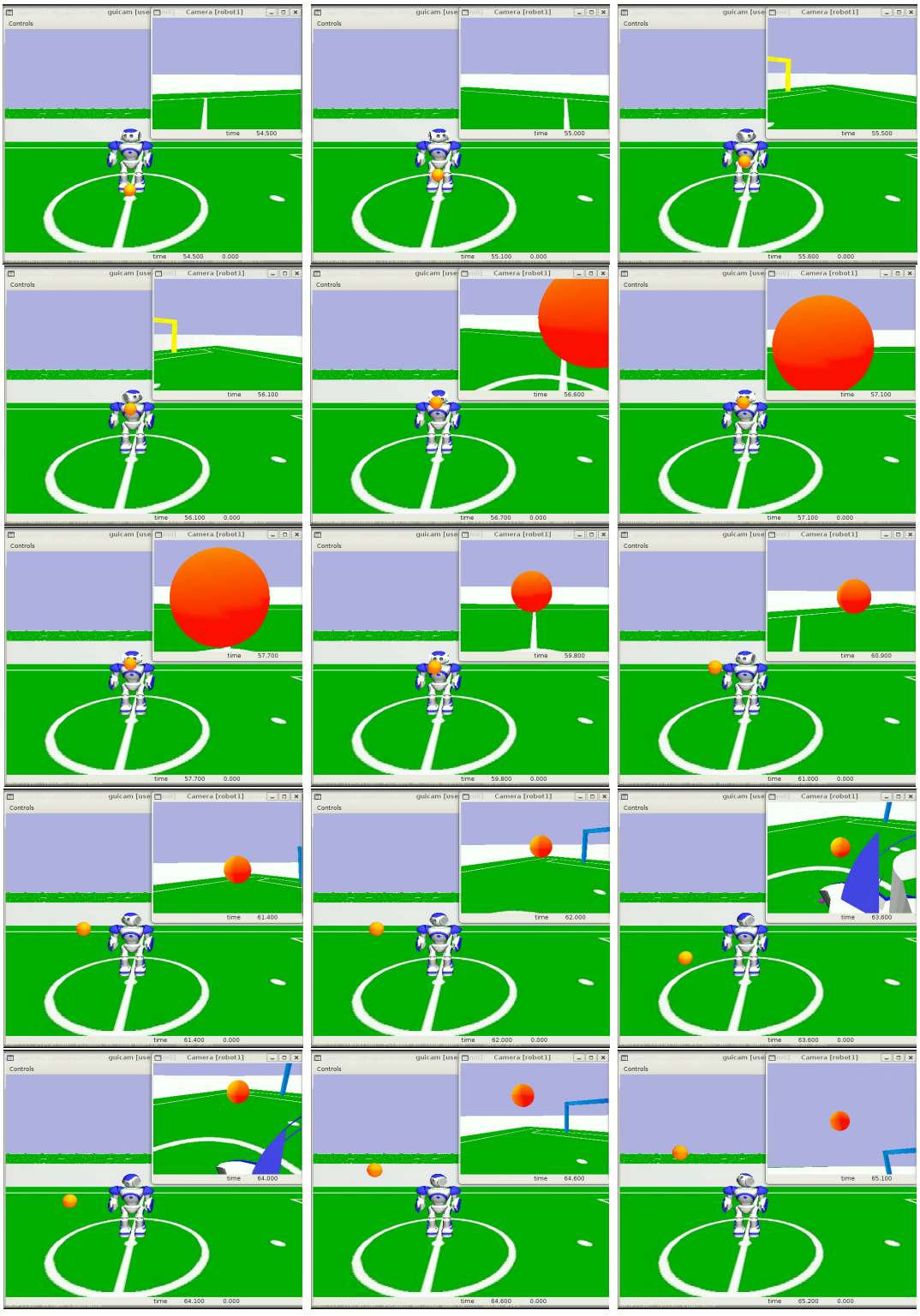
Con este componente nos centramos en validar el correcto funcionamiento de la cámara y el cuello del robot Nao simulado. El objetivo es que el robot Nao busque la pelota y, una vez que la ha encontrado la siga con la mirada, intentando que la pelota este siempre lo más cerca posible del centro de la imagen capturada con la cámara.

El esquema Faceball recibe como entrada una imagen en formato RGB, con una resolución de 320x240 píxeles. Una vez que tenemos a nuestra disposición la imagen en formato RGB procedente de la cámara, pasamos a analizarla. El espacio sobre el que se realiza el filtro de color es el espacio HSI. Este espacio posee tres componentes: la componente H, que es el Matiz del pixel, o lo que es lo mismo, el color en sí; la componente S, que es la Saturación, es decir, identifica la claridad del color; y la componente I que es la Intensidad del color o luminosidad. Al aplicar un filtro en este espacio obtenemos una mayor robustez ante cambios de luminosidad, ya que podemos ignorar la componente I. De esta manera, lo primero que efectúa el esquema `faceball` es una conversión de espacio RGB a espacio HSI siguiendo las ecuaciones:

$$H = arccos\left(\frac{\frac{(R-G)+(R-B)}{2}}{\sqrt{(R-G)^2+(R-B)(G-B)}}\right)$$

$$S = 1 - \left(\frac{3}{(R+G+B)}\right)[min(R,G,B)]$$

[1] http://jde.gsyc.es/index.php/Jbermejo-pfc-itis



(a)

***Figura 5.1:*** *Secuencia de la aplicacion Faceball(a).*

$$I = \frac{1}{3}(R + G + B)$$

Donde R corresponde a la componente roja, G a la componente verde, y B a la componente azul de la imagen origen en formato RGB. Así como la función *min(R,G,B)* devuelve el mínimo valor de las tres componentes.

Una vez tenemos la imagen en el espacio HSI, definimos un rango de valores correpondientes a cada una de las componentes del HSI. Este rango de valores vendrá dado por un $Imaximo$, un $Iminimo$, un $Hmaximo$ y un $Hminimo$, así como un $Smaximo$ y un $Sminimo$ definidos para el color de la pelota. Para realizar el filtrado, se analiza píxel a píxel la imagen, comprobando que los valor de matiz, de saturación y de luminosidad de cada pixel entra dentro del rango de valores H, S , I establecidos. Si cumple estas condiciones marcamos el pixel, así, al final del proceso tenemos una imagen de dimensión y formato igual al de la imagen de entrada, pero con una serie de pixeles "marcados" que identifican a la pelota. Esta primera parte podemos ver como está implementada en la lista 5.1.

El esquema `Faceball` además de identificar la pelota, calcula su posición en la imagen. Para ello calcula el centro de masas de los pixeles de la pelota, que vendrá definido por un pixel de coordenadas x e y en la imagen, obtenidas de la siguiente manera:

$$\widehat{x} = \frac{\sum_{i=1}^{P} X_i}{P}$$

$$\widehat{y} = \frac{\sum_{i=1}^{P} Y_i}{P}$$

siendo P el total de pixeles marcados. Para realizar esto debemos introducir el código de la lista 5.2.

Así, ya tenemos detectada la pelota y en el cuadrante en el que se encuentra. El siguiente paso es mover el cuello del robot en las direcciones *pan* y *tilt* para centrar la pelota en la imagen. El cuello se moverá siempre y cuando supere una distancia mínima de píxeles que le marcamos. Esta condición se utiliza ya que es muy difícil que la pelota este exactamente en el centro de la imagen, con lo que con esta distancia mínima lo que evitamos es que esté continuamente en movimiento cuando el movimiento de la pelota ha sido mínimo.

Como ya hemos indicado, gracias a este componente observamos el correcto funcionamiento tanto de la recepción de las imágenes de la cámara como del movimiento del cuello del robot Nao.

```
1
2   for(i=0;i< SIFNTSC_ROWS*SIFNTSC_COLUMNS; i++)
3   {
4
5       r = (float)(unsigned int)(unsigned char)(*mycolorA)[i*3+2];
6       g = (float)(unsigned int)(unsigned char)(*mycolorA)[i*3+1];
7       b = (float)(unsigned int)(unsigned char)(*mycolorA)[i*3];
8
9       myHSV = (struct HSV*) RGB2HSV_getHSV( (int)r,(int)g,(int)b);
10      H=myHSV->H;
11      S=myHSV->S;
12      I=myHSV->V;
13
14      if((I<=i_max)&&(I>=i_min)&&(S >= s_min)
15          && (S <= s_max) && (H >= h_min) && (H <= h_max) )
16      {
17          pixeles++;
18          data_filter .x += pixel_x;
19          data_filter .y += pixel_y;
20          pasa_filtro =1;
21      }
22
23
24      /* Hacemos una conversion para simular que estamos
25      sobre un array dimensional y saber en que pixel (x,y) estamos */
26      if (pixel_x==320)
27      {
28          pixel_x=1;
29          pixel_y++;
30          if ( pasa_filtro ==1){
31              num_lineas++;
32          }
33          pasa_filtro =0;
34
35      } else{
36          pixel_x++;
37      }
38  }
```

***Listing 5.1:*** *Código filtrado de color para la pelota*

```
1
2   /* hallamos la media de los puntos */
3   data_filter.x = data_filter.x / data_filter.pixeles;
4   data_filter.y = data_filter.y / data_filter.pixeles;
5
6   /* hallamos las coordenadas reales para el centro (160,120)  */
7   data_filter.x = data_filter.x - 160;
8   data_filter.y = data_filter.y - 120;
9
10  /* hallamos la distancia entre el centro y el punto medio */
11  data_filter.distancia = sqrt((sqr(data_filter.x)+sqr(data_filter.y)));
12
13  /* miramos en que cuadrante se encuentra */
14  if (data_filter.x > 0)
15      if (data_filter.y > 0)
16          data_filter.cuadrante=1;
17      else
18          data_filter.cuadrante=3;
19  else
20      if (data_filter.y > 0)
21          data_filter.cuadrante=2;
22      else
23          data_filter.cuadrante=4;
24
25  if ((data_filter.x==0) && (data_filter.y==0))
26      data_filter.cuadrante=0;
27  printf("cuadrante= %d\n",data_filter.cuadrante);
28  pantilt_iteration();
```

***Listing 5.2:*** *Código cálculo de posición de la pelota en la imágen*

## 5.2. Baile

El segundo experimento lo realizamos para ver que nuestro robot Nao simulado cumple con los objetivos que nos marcamos para este proyecto. Este experimento consiste en la realizacion de un baile con el cual podamos observar como es capaz de mover correctamente todas las articulaciones que posee nuestro robot.

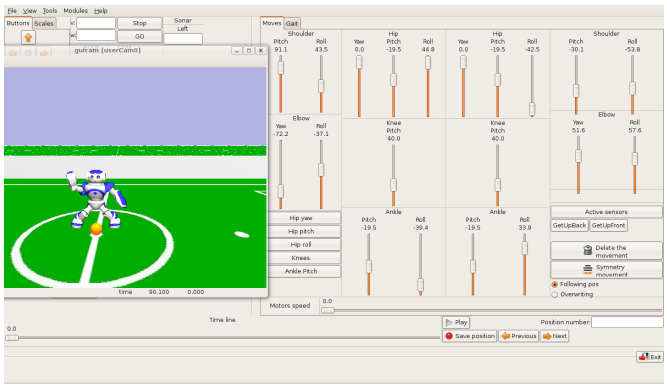
Con este experimento comprobamos la capacidad de movilidad del robot Nao materializado en Gazebo, además tambien demostramos la compatibilidad con aplicaciones anteriores de JDERobot.

Para la realización de este experimento hacemos uso del esquema **NaoOperator**, desarrollado por Francisco Miguel Rivas. Este esquema fue desarrollado, junto con el driver **naobody**, para poder teleoperar el robot Nao real. La utilidad básica como controlador de un robot es poder manejar cada uno de los dispositivos del robot, hacer que ande, controlar la cámara, mover los motores, obtener diferentes valores de los sensores...

Además de esta tarea de manejo básico del robot, el esquema **NaoOperator** también incorpora algunas utilidades como el generador de posturas o movimientos, capacidad para variar los parametros en la forma de andar... Este esquema ha sido probado con éxito tanto en el robot real como en el robot simulado en Webots.

Uno de lo módulos mas útiles que incorpora el NaoOperator y del cual vamos a hacer uso en nuestro experimento, es un módulo para crear movimientos con acceso completo a todo los motores del robot. La idea del generador de movimientos es bastante sencilla: guardamos una serie de posiciones estáticas con el robot usando el *gui* para crearlo y le asignamos un cierto valor de tiempo a cada posición. Una vez guardadas todas las posiciones estáticas el NaoOperator se encarga de generar un movimiento pasando por cada una de nuestras posiciones fijas. Con este módulo podemos crear cualquier tipo de movimientos (levantarse, disparos, andar...).

Como ya hemos dicho anteriormente el generador de movimientos se basa en posiciones fijas(figura 5.2(a)), por lo que lo primero será crear una posición y



(a)

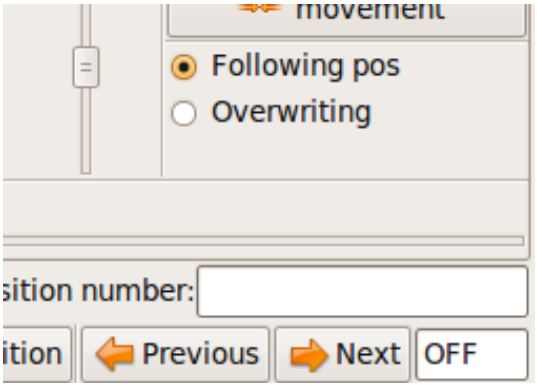
***Figura 5.2:*** *Posición fija*

asignarle un tiempo. En nuestro caso uno de los pasos del baile que va a realizar.

Para ello tenemos que seleccionar la pestaña **Moves** y variando los valores de los motores creamos la posición que queramos que realice el robot Nao. Una vez creada la posición tenemos que asignarle un tiempo y despues guardardarlo.

Tendremos que repetir este paso las veces necesarias para crear el movimiento deseado.

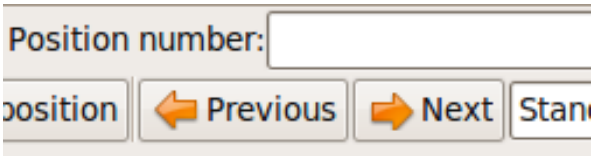
Al guardar una posición tenemos dos opciones de guardado, o bien la guardamos en una posición nueva o bien modificamos la posición actual. Esta opción la podemos elegir mediante los radios que vemos en la figura 5.3(a)




(a)

***Figura 5.3:*** *Selección de radio*

Con esto podremos modificar movimientos ya creados ya que con el menu que vemos en la figura 5.4(a), podemos navegar por las diferentes posiciones de un movimiento y modificarlo si es conveniente.



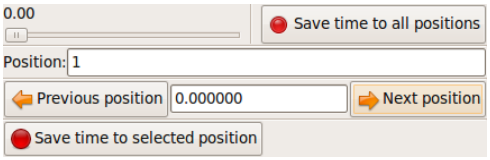

(a)

***Figura 5.4:*** *Selección de posición*

Algunos de las herramientas útiles que incorpora el esquema NaoOperator a la hora de crear movimientos y de las cuales hemos hecho uso para crear nuestro baile son:

- **Modificador de los tiempos de las posiciones**

  A veces es necesario poder modificar los tiempos de la posiciones y esto no lo podemos hacer directamente con el generador de movimientos por lo que tendremos que utilizar esta herramienta. Usando el **Modificador de tiempos** podemos variar todos los tiempos a la vez con un intervalo común, es decir, que cada posición tenga un mismo tiempo (1 segundo entre posición y posición) o bien variar el tiempo de posiciones independientemente. (figura 5.5(a))



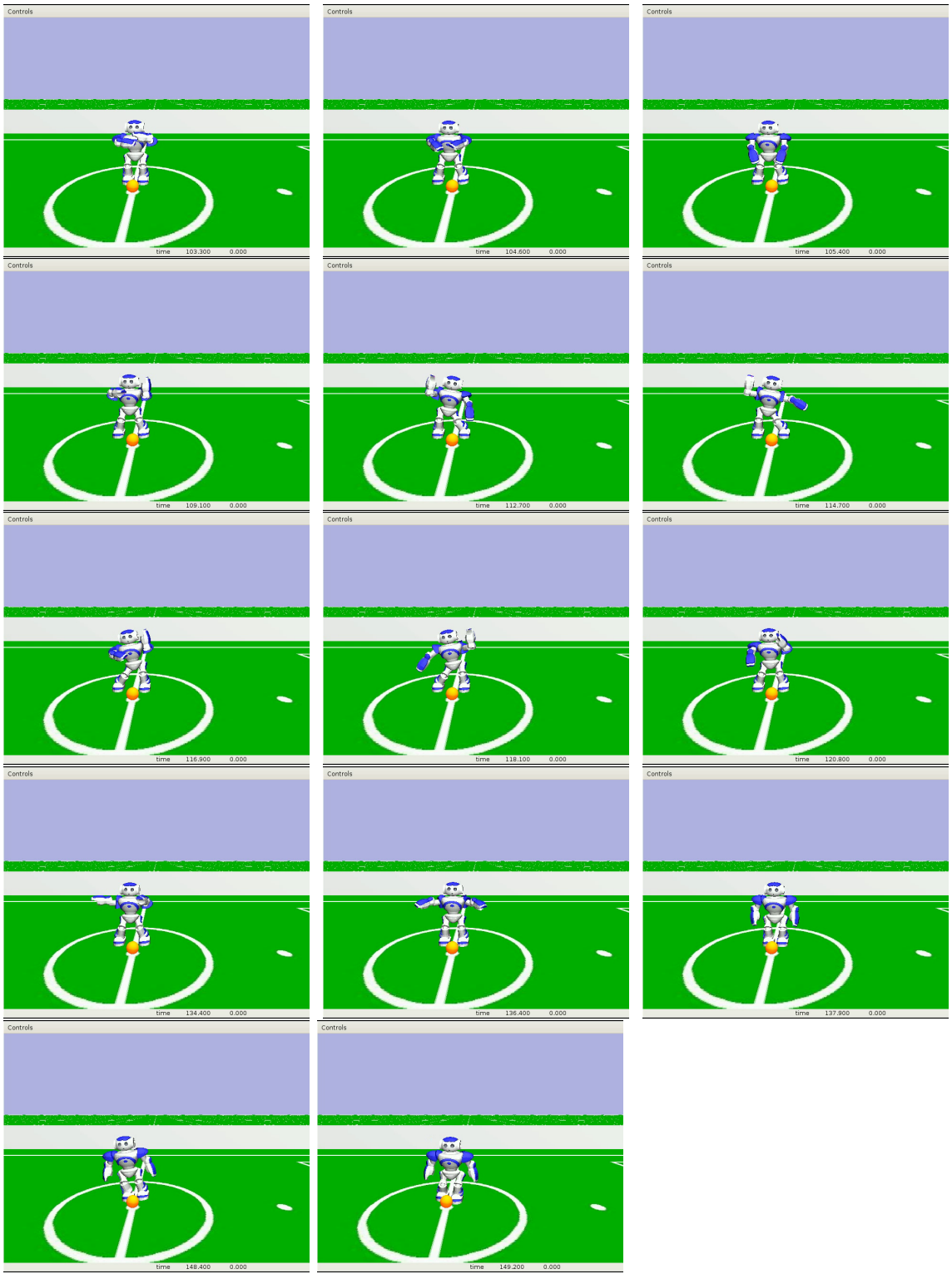

(a)

***Figura 5.5:*** *Modificador de los tiempos de las posiciones*

- **Modificador de máximos y mínimos**

  Una vez generado un movimiento es posible que necesitemos que alguno de los motores o todos no generen movimientos tan amplios o tan simples y queremos mantener el mismo movimiento pero con unos rangos diferentes. Esto lo podemos hacer directamente con esta herramienta. Una vez generado el movimiento o cargado uno desde un fichero activamos el **Modificador de máximos y mínimos** y simplemente con variar los rangos que nos aparecen en la herramienta variaran todos los valores de las posiciones con lo que conseguimos cambiar el movimiento final.

Gracias a este esquema conseguimos que nuestro robot Nao simulado en Gazebo realizase un pequeño baile, con el que podemos comprobar como es capaz de mover todas y cada una de las articulaciones que posee. Como se puede ver en los videos del Mediawiki, y en la figura 5.6(a) el robot mueve las articulaciones de los hombros, la de los codos, incluso es capaz de andar y hasta de mover las caderas mientras realiza algun movimiento con los brazos.



(a)

***Figura 5.6:*** *Secuencia de la aplicacion Baile(a).*

# Capítulo 6

# Conclusiones y trabajos futuros

En los capítulos anteriores hemos descrito el problema que abordamos en este proyecto, en este capitulo se resumen las conclusiones principales que hemos alcanzado, y la solución propuesta junto con un conjunto de experimentos que la validan. Además, se proponen un conjunto de líneas útiles de investigación que pueden continuar este trabajo, cerrando así la presente memoria.

## 6.1. Conclusiones

Como conclusión global destacamos que se han cumplido satisfactoriamente los objetivos expuestos en el capítulo 2 de esta memoria, así como ciertos requisitos que tenían implicitos y que condicionaban la solución que debíamos obtener:

1. El primero de los subobjetivos a alcanzar era la creación de un modelo de robot Humanoide en el entorno de simulación Gazebo. Este primer subobjetivo fue resuelto creando el robot mediante las figuras geométricas básicas que nos proporciona Gazebo para la construcción de modelos. Se utilizaron hexaedros rectangulares para la mayoría de partes del cuerpo, y una esfera para la simulación de la cabeza. Fue desarrollado por partes, comenzando por el tronco y las extremidades superiores y finalizando por añadir las extremidades inferiores.

   Cuando se fueron construyendo cada una de las partes del modelo humanoide simulado se iban juntando con las ya creadas mediante las articulaciones que proporciona Gazebo. Todas las partes del modelo fueron creadas basándonos en las medidas y pesos que tiene el robot Nao real, validando así uno de los requisitos que nos marcamos. Al desarrollarlo sobre la plataforma de simulación de robots Gazebo, concretamente en su versión 0.7, resolvemos también el segundo de nuestros requisitos.

2. El siguiente subobjetivo era darle al robot humanoide simulado en Gazebo una apariencia más cercana al robot Nao real. Para ello utilizamos el software de diseño 3D Blender para crear cada una de las partes del robot en 3D. Los archivos 3D fueron añadidos al modelo al cargar nuestro "Mundo" en Gazebo. De este modo, al cargarle una "piel" sobre la base construida mediante figuras

geométricas básicas dotamos a nuestro robot Nao simulado de una apariencia más real. Con este subobjetivo además, resolvemos el segundo de los requisitos propuestos, que consistía en construir la piel del robot simulado con el software de diseño 3D Blender.

3. El tercero de los subobjetivos consistía en desarrollar los componentes necesarios para dar soporte al control y manejo de nuestro robot Nao simulado. Como ya mencionamos en los capítulos 3 y 4 de esta memoria el driver gazebo incluido en la versión 4.3 de JDERobot no soporta el control de un robot humanoide, por lo que fue necesario desarrollar en este proyecto un nuevo driver. Este driver consta de un thread que va preguntando a las articulaciones en que posición se encuentran en cada momento de la simulación. Además, en la cabeza también tenemos una cámara, a la cual podemos acceder mediante VarcolorA y así poder recoger y mostrar las imágenes obtenidas.

   Con este driver podemos manejar y controlar cualquier movimiento que realice el robot Nao simulado. El nuevo driver Gazebo desarrollado se ha incorporado a la versión oficial de JDERobot y constituye un aporte lateral de este proyecto a la infraestructura de la plataforma. Gracias a este *driver* obtenemos una compatibilidad total con las aplicaciones de JDERobot creadas anteriormente, así podremos reutilizar aplicaciones que ya habían sido creadas y probadas con éxito tanto para el robot real como para el robot simulado con Webots.

Con relación a los conocimientos adquiridos durante el desarrollo del proyecto fin de carrera, se debe remarcar la gran cantidad de herramientas y técnicas aprendidas, lo que incluye la plataforma JDE como entorno de desarrollo sobre el sistema operativo Linux, el lenguaje de programación C, el simulador 3D Gazebo, la aplicación de diseño gráfico Blender, el sistema de control de versiones SVN y muchas otras herramientas muy importantes en lo que a desarrollo software se refiere, como Latex, etc.

Además, gracias a la realización de este proyecto de fin de carrera colabooramos con la comunidad internacional de Gazebo, pues en breve se propondrá la inclusión en la versión oficial de Gazebo el soporte para el robot Nao. Incluyendo de esta forma en un software de simulación gratuito uno de los robots más utilizados en la actualidad debido a su participación en la competición internacional de la Robocup.

## 6.2. Trabajos futuros

Este proyecto fin de carrera abre un abanico de posibles líneas de trabajo futuras:

En primer lugar podemos destacar el hecho de que actualmente el robot Nao simulado está soportado por Gazebo en su versión 0.7, por lo que la primera línea de trabajo podría ser actualizarlo para que estuviese soportado en la última versión de Gazebo, que actualmente es la 0.10.

Otra posible línea de investigación puede ser añadirle al robot simulado el resto de sensores que incorpora el robot Nao, como la otra cámara de la cabeza, ya que actualmente solo incorpora una de ellas.

También sería posible añadirle otros sensores como el FSR, los 4 sonar's del pecho, etc.

En un futuro, también cabría la posibilidad de modificar el humanoide propuesto, el robot Nao de Aldebaran, para que se pareciese a otro robot Humanoide de similares características. Cambiando el tamaño y las masas de las diferentes partes del modelo, podriamos crear otro modelo de robot distinto.

# Bibliografía


[Cañas Plaza, 2008] Jose María Cañas Plaza. Manual de programación de robots con jde. *Universidad Rey Juan Carlos*, noviembre 2008.

[Gates, 2007] Bill Gates. A robot in every home. *Scientyfic American*, 2007.

[González Morcillo, 2005] Carlos González Morcillo. Blender y yafray, diseño gráfico 3d con software libre. *Escuela superior de Informática - Universidad de Castilla-La Mancha*, 2005.

[Koenig, 2007] Nate Koenig. Gazebo: The instant expert's guide. *Player summer school on cognitive Robotics*, 2007.

[León, ] Luis Alvarez León. Tutorial de latex. *Universidad de Las Palmas de Gran Canaria*.

[Martinez Gil, 2003] Juan José Martinez Gil. Equipo de fútbol con jde para la liga simulada robocup. *Proyecto Fin de Carrera. ITIS - Universidad Rey Juan Carlos*, 2003.

[Muelas Sahagún, 2008] David Muelas Sahagún. Visualizador 3d interactivo para laboratorios de análisis de marcha. *Proyecto Fin de Carrera. ITIS - Universidad Rey Juan Carlos*, 2008.

[P. Gerkey and Howard, 2003] Brian P. Gerkey and Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the international conference on Advanced Robotics.*, pages 1–36, 2003.

[Perdices García, 2009a] Eduardo Perdices García. Autolocalización visual en la robocup basada en detección de porterías 3d. *Proyecto fin de carrera - Ingeniería en informática superior - Universidad Rey Juan Carlos*, 2009.

[Perdices García, 2009b] Eduardo Perdices García. Autolocalización visual en la robocup basada en detección de porterías 3d. *Proyecto fin de carrera - Ingeniería en informática superior - Universidad Rey Juan Carlos*, 2009.

[Plaza, 2003] José María Cañas Plaza. Jerarquía dinámica de esquemas para la generación de comportamiento autónomo. *Tesis doctoral, Universidad Politécnica de Madrid*, 2003.

[Plaza, 2004] José María Cañas Plaza. Manual de programación de robots con jde. *URJC*, 2004.

[RoboCup, 2009] RoboCup. Robocup standard platform league (nao) rule book. http://www.tzi.de/spl/pub/Website/Downloads/Rules2009.pdf, Mayo 2009.