INGENIERÍA INFORMÁTICA

Escuela Técnica Superior de Ingeniería Informática

Curso Académico 2008/2009

Proyecto de Fin de Carrera

Autolocalización visual en la RoboCup basada en detección de porterías 3D

Autor: Eduardo Perdices García

Tutor: José María Cañas Plaza

A mi familia y a todos mis amigos.

Gracias.

Agradecimientos

Quiero dar las gracias a todos los miembros del Grupo de Robótica de la Universidad Rey Juan Carlos por su apoyo y colaboración. En especial a Julio y Gonzalo por haberme ayudado en numerosas ocasiones durante el desarrollo del proyecto.

También quisiera agradecer a José María todo el trabajo, dedicación y ayuda que me ha prestado durante todos estos meses de trabajo.

Como no podía ser de otra forma quisiera dar las gracias a toda la gente de la universidad que ha pasado por la carrera conmigo durante todos estos años, a Jafet, Santiago, Daniel, Mario, David, Carlos, Jose, y otros muchos más.

Además quisiera dar las gracias a mis mejores amigos por haber estado siempre a mi lado en los buenos y malos momento, a Diego, Javier, Carolina, Marina y en especial a Estefanía, sin olvidar a otros muchos que también merecerían ser nombrados.

A todos, muchas gracias!

Resumen

La autolocalización de robots en entornos conocidos es hoy en día uno de los retos más importantes dentro del campo de la robótica. A partir de los distintos sensores de los que cuenta el robot, como sensores de ultrasonido, sensores láser o cámaras, el robot tiene que estimar su posición en el entorno de trabajo. En este proyecto hemos realizado una serie de algoritmos de autolocalización para localizar a un robot humanoide Nao dentro del campo de fútbol de liga SPL de la RoboCup, utilizando únicamente su cámara.

Se han utilizado las porterías completas del campo para realizar esa localización, detectando mediante visión artificial estas porterías en las imágenes recibidas desde la cámara, y aplicando unas restricciones geométricas (toros, esferas, arco capaz, rayos proyectivos) para estimar la posición de la cámara en 3D. Se ha planteado la localización desde dos puntos de vista distintos, primero desde una única imagen y después utilizando modelos probabilísticos y la regla de Bayes para combinar varias observaciones.

Para el desarrollo de la aplicación se han utilizado los lenguajes de programación C y C++, bajo la plataforma de desarrollo JdeRobot. Esta plataforma nos ha permitido reutilizar componentes desarrollados en otros proyectos de una forma sencilla. Además, a lo largo del proyecto se ha hecho uso de otras tecnologías como OpenGL, OpenCV o GTK.

Índice general

Índice de figuras

Capítulo 1

Introducción

En este capítulo vamos a presentar los elementos más importantes del contexto de nuestro proyecto, que está relacionado con la robótica, la visión artificial y la autolocalización. También haremos una descripción detallada de la competición en la que está enfocado el proyecto, la RoboCup.

1.1. Robótica

Se define la robótica como la ciencia y la tecnología de los robots, combinando en ella varias disciplinas como la mecánica, la informática, la electrónica y la ingeniería artificial, que hacen posible el diseño hardware y software del Robot. Un robot, es un sistema electromecánico que utiliza una serie de elementos hardware, como por ejemplo, actuadores, sensores y procesadores, los cuales rigen su comportamiento por un software programable que le dan la vida y la inteligencia al robot.

El primer robot programable se construyó en 1961, fue conocido como Unimate y servía para levantar piezas industriales a altas temperaturas, sin embargo no sería hasta la década de los 70 cuando se comenzase a desarrollar del todo esta tecnología, creándose en 1973 el primer robot con 6 ejes electromecánicos (conocido como Famulus) y en 1975 el primer brazo mecánico programable (PUMA).

A partir de los años 80 se comenzaron a utilizar en masa este tipo de robots, ya que proporcionaban una alta rapidez y precisión, y se utilizaron sobre todo para la fabricación de coches (figura 1.1(a)), empaquetamiento de comida y otros bienes, y la producción de placas de circuitos impresos.

Desde entonces se han realizado grandes avances en este campo, existiendo actualmente robots para todo tipo de propósitos, siendo ampliamente utilizados sobre todo en tareas industriales, donde realizan el trabajo de una forma mucho más precisa y barata que los humanos. También se utilizan robots en campos como el rescate de personas y la localización de minas, salvando muchas vidas humanas, además de utilizarse en diversos campos de la medicina, como la cirugía de alta precisión.

Además de los mencionados anteriormente, la robótica también se utiliza en un amplio espectro de aplicaciones, como la ayuda de las tareas del hogar, vigilancia, educación, o con fines militares.

Uno de los campos donde destaca la utilización de robots es en las misiones espaciales a otros planetas, que nos sirven para explorar la superficie de éstos sin la necesidad de la presencia de los humanos.

En los últimos años se han desarrollado diferentes robots especializados en la movilidad en un terreno, ya sea conocido o desconocido, para ello, los robots hacen uso de distintos sensores que les permiten captar la información del exterior y a partir de ellos son capaces de localizar su posición en un escenario conocido, o de navegar a un destino solicitado.

Un ejemplo de este tipo de retos, es la *Urban Challenge* [1], donde se realiza una competición con vehículos totalmente autónomos capaces de moverse entre el tráfico de una ciudad para alcanzar una serie de objetivos, y en la cuál es muy importante para el robot conocer dónde está.

Otro de los robots que destaca por su movilidad en superficies es *Roomba*, un robot de limpieza automático, que para utilizarlo sólo necesitamos colocarlo en el suelo y él automáticamente se encarga de recorrer toda la superficie del lugar donde nos encontremos.

También están proliferando en la actualidad muchos robots humanoides, que intentan simular la forma de andar de los humanos e interactúan con éstos utilizando multitud de sensores. Un ejemplo de este tipo de robots, es el robot *Asimo*, desarrollado por Honda y cuya imagen puede verse en la figura 1.1(b).

[1] http://www.darpa.mil/grandchallenge/index.asp

(a)

(b)

Figura 1.1: Robots industriales fabricando coches (a). Robot *Asimo* (b).

1.2. Visión artificial en robótica

Para que el robot tenga información del entorno en el que se encuentra, se pueden utilizar diversas fuentes de información, como sensores de ultrasonido, sensores láser, etc, que nos presentan información sobre distancias a objetos, o se pueden utilizar cámaras, con las que podemos extraer cierta información del mundo mediante la visión artificial.

La visión artificial es un campo de la inteligencia artificial que pretende obtener información del mundo a partir de una imagen, que normalmente vendrá dada en forma de matriz numérica. La información relevante que se puede obtener a partir de la imagen puede ser el reconocimiento de objetos, la recreación en 3D de la escena que se observa, el seguimiento de una persona u objeto, etc.

El inicio de la visión artificial se produjo en 1961 por parte de Larry Roberts, quien creó un programa que podía ver una estructura de bloques, analizar su contenido y reproducirla desde otra perspectiva, utilizando para ello una cámara y procesando la imagen desde el computador. Sin embargo para obtener este resultado las condiciones de la prueba estaban muy limitadas, por ello otros muchos científicos trataron de solucionar el problema de conectar una cámara a un computador y hacer que éste describiese lo que veía. Finalmente, los científicos de la época se dieron cuenta de que esta tarea no era tan sencilla de realizar, por lo que se abrió un amplio campo investigación, que tomó el nombre de visión artificial, con el que se pretende conseguir lo anteriormente descrito para dar un

gran paso en la inteligencia artificial.

Así, se intenta que el computador sea capaz de reconocer en una imagen distintos objetos al igual que los humanos lo hacemos con nuestra visión. Sin embargo, se ha demostrado que este problema es muy complejo y que algo que para nosotros puede resultar automático, puede tardarse años en resolver para una máquina.

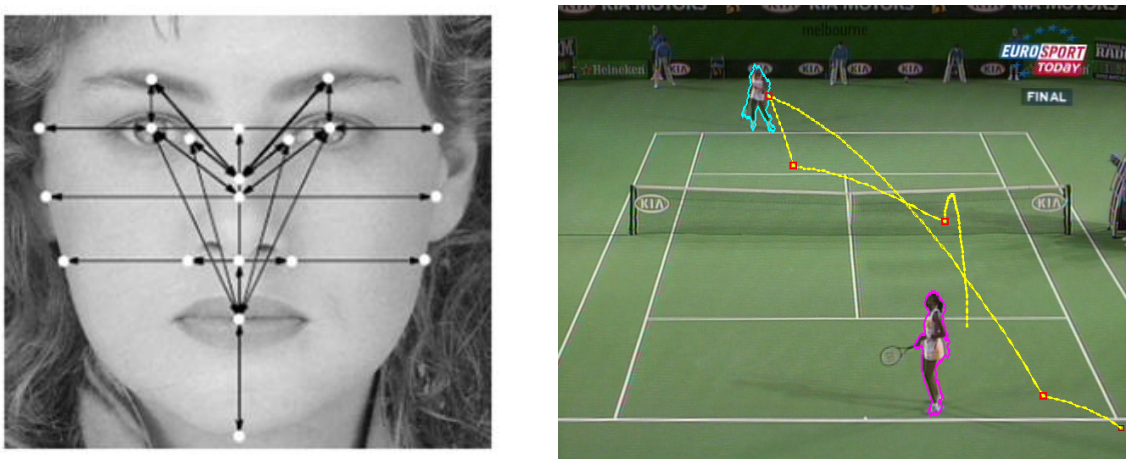
Por otra parte, a pesar del alto precio computacional que se paga al utilizar las cámaras como fuente de información, si se consigue analizar correctamente la imagen, es posible extraer mucha información que no podría obtenerse con otro tipo de sensores.

A comienzos de los años 90 comenzaron a aparecer ordenadores capaces de procesar lo suficientemente rápido imágenes, por lo que comenzaron a dividirse los posibles problemas de la visión artificial en otros más específicos que pudiesen resolverse. Actualmente se utiliza en muchos procesos científicos, militares o industriales, para el reconocimiento de objetos o en el seguimiento de éstos:

- Reconocimiento de objetos: A partir de una serie de características de un objeto, como puede ser su forma, su color o cualquier otro patrón, pueden compararse mediante algoritmos estas características con la imagen obtenida, para determinar si se encuentra algún objeto que siga este patrón (Figura 1.2(a)).
- Seguimiento de objetos: Una vez detectado un objeto, podemos realizar tareas de seguimiento de éste teniendo en cuenta que el objeto tiene 6 posibles grados de libertad, 3 de traslación y 3 de rotación. El seguimiento del objeto puede realizarse utilizando como fuente sus propiedades (bordes, esquinas, texturas, etc) o bien seleccionando una serie de puntos característicos del objeto y realizando únicamente el seguimiento de estos puntos. El seguimiento de objetos puede tener múltiples finalidades, desde la vigilancia en centros comerciales hasta la ayuda en algunos deportes (Figura 1.2(b)).

A pesar de que obtener información mediante visión artificial tiene una gran complejidad, las cámaras son el sensor más utilizado en los robots para percibir lo que les rodea. Esto es debido a que es un sensor muy barato comparado con el resto, y además la información que podemos obtener a partir de él es muy grande.

Algunos de los robots que vimos en la sección anterior utilizan las cámaras como sensor principal, éste es el caso de los vehículos autónomos utilizados en la *Urban Challenge*, que necesitan la información que les proporcionan las cámaras para poder ver las señales de tráfico y evitar a otros vehículos.



(a) (b)

Figura 1.2: Reconocimiento de cara en imagen (a). Seguimiento de pelota durante un partido de tenis (b).

1.3. RoboCup

La RoboCup [2] es un proyecto a nivel internacional para promover la inteligencia artificial, la robótica y otros campos relacionados. Se trata de promocionar la investigación sobre robots e IA proporcionando un problema estándar donde un amplio abanico de tecnologías pueden ser integradas y examinadas. La meta final de la RoboCup es: "Para el año 2050, desarrollar un equipo de fútbol de robots humanoides totalmente autónomos capaces de jugar y ganar 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 o, el campo en el que está centrado nuestro proyecto, la autolocalización.

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.

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

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 grandes competiciones:

- RoboCupSoccer: Proyecto principal del que ya hemos hablado, donde se realizan competiciones de fútbol entre robots autónomos.
- RoboCupRescue: Se pone a prueba a los robots en tareas de búsqueda y salvamento en terrenos desfavorables. En esta ocasión, los robots pueden ser tanto autónomos como guiados por control remoto (Figura 1.3(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 1.3(b)).

(a) (b)

Figura 1.3: Robot de rescate para la RoboCupRescue (a). Robot que compite en una de las ligas de la RoboCup@Home (b).

Dentro de cada competición existen diversas categorías, a continuación vamos a explicar las distintas ligas existentes para la RoboCupSoccer, ya que es la utilizada en nuestro proyecto:

- Liga de simulación: No existen robots físicos, por lo que sólo se enfrentan robots en simulaciones virtuales.

- 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 1.4(a)) y que cuenten 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, siendo la visión su sensor principal.

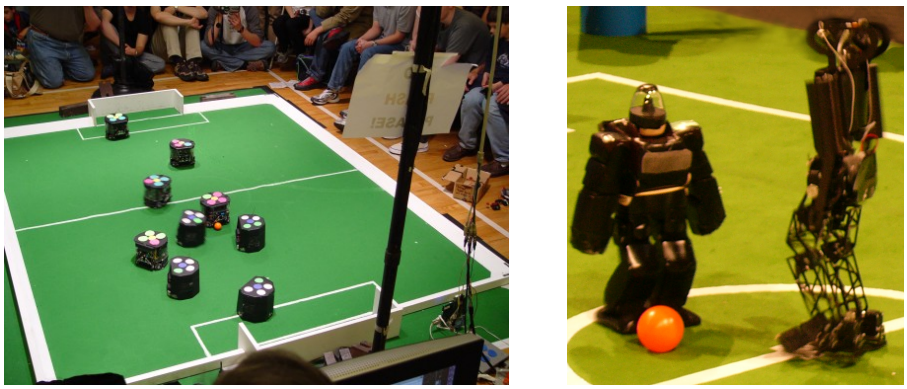
- Plataforma estándar: Categoría en el que todos los participantes utilizan el mismo robot y sólo se centran en el desarrollo del software de éste. Actualmente se utiliza el robot Nao de Aldebaran Robotics (1.5), aunque hasta 2007 se utilizó el robot Aibo de Sony. Al igual que en la liga anterior, los 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 para 2050.

- 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 1.4(b)).

En las ligas que acabamos de describir, uno de los factores determinantes es saber qué hacer cuando percibimos la pelota, puesto que la actuación del robot deberá ser distinta cuando se encuentre en su propia área (intentará despejar) que cuando esté en el área contraria (intentará tirar a portería). Por ello, uno de los problemas más importantes que se plantean es la localización dentro del campo, que se debe realizar utilizando solamente la cámara de la que disponen.

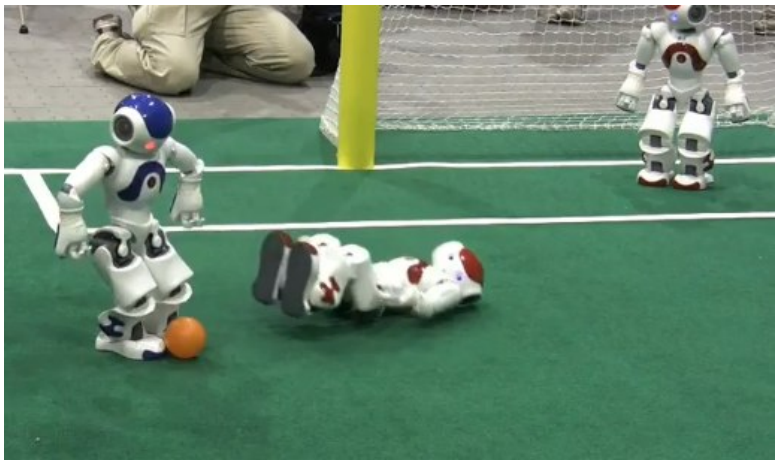
Desde el año 2001, el grupo de Robótica de la URJC trabaja en proyectos relacionados con la RoboCup, estudiando cómo deberían comportarse los robots dentro del campo. Así, pueden destacarse algunos proyecto como [Alvarez Rey, 2001], quien desarrollo un comportamiento para un equipo de la liga de simulación utilizando lógica borrosa, [Peño, 2003], cuyo proyecto consistió en generar el comportamiento 'sacar de banda' utilizando un robot EyeBot, [Martinez Gil, 2003], quien programó un equipo de fútbol software capaz de jugar en la liga simulada de la RoboCup, o [Crespo, 2003], que planteó el problema de la localización en el campo de la RoboCup con los robots Eyebot.

Sin embargo, aún no se había realizado ningún proyecto con los robots utilizados actualmente en la plataforma estándar de la RoboCup, los robots humanoides Nao. Por ello, partiendo de los proyectos anteriores y después de comprobar la importancia de la localización dentro del campo, decidimos enfocar nuestro proyecto en la autolocalización de robots dentro del campo de la RoboCup.



(a) (b)

Figura 1.4: Liga de robots de tamaño pequeño (a). Liga de humanoides (b).



(a)

Figura 1.5: Robots Nao durante la competición de la RoboCup 2008.

1.4. Técnicas de Autolocalización

La localización consiste en determinar con cierta certeza en qué posición se encuentra el robot en un momento concreto a partir de su información sensorial. Para ello, un robot puede utilizar distintos mecanismos, puede hacer uso de la odometría, utilizar sensores especializados como los GPS, o usar otros sensores, como las cámaras.

- Odometría: Consiste en determinar la posición del robot conociendo el movimiento de los motores que dispone y de su posición anterior. Por ejemplo, en el caso de un robot que se mueva con ruedas, puede determinarse la nueva posición conociendo cuantas vueltas ha dado cada rueda y las dimensiones de éstas. En el caso de robots humanoides o en robots con patas, los cálculos son mucho más complejos puesto que hay que tener en cuenta un gran número de actuadores que se mueven simultáneamente.

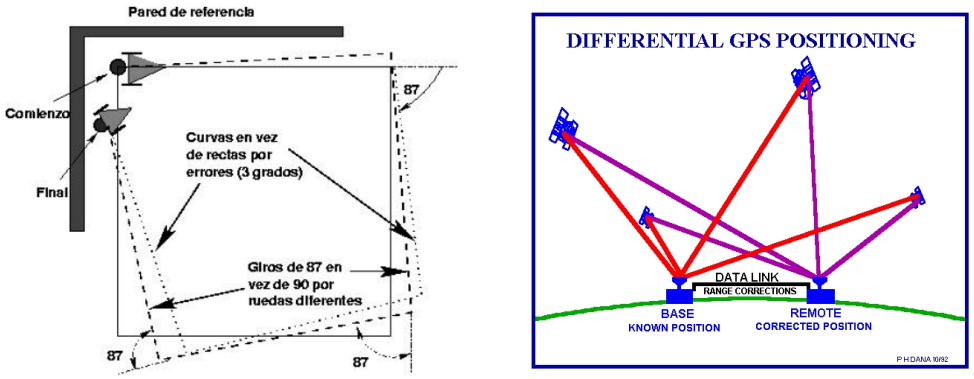
  Además, esta técnica es poco fiable, ya que puede haber errores de cálculo debido a deslizamientos, valores poco precisos de los actuadores, etc. Un ejemplo de lo anterior puede verse en la figura 1.6(a), donde un robot intenta moverse haciendo un cuadrado pero varía su trayectoria por errores de precisión.

- GPS: El sensor GPS recibe señales de satélites que le permiten determinar la posición del robot en cualquier parte del mundo, con una precisión de pocos metros. El sistema GPS se compone de 27 satélites que orbitan alrededor de la tierra con trayectorias sincronizadas. Para determinar la posición, se recibe la señal de al menos 3 satélites, que indican la posición y la hora de cada uno de ellos, con estos datos y sincronizando el retraso de las señales puede calcular su posición por triangulación.

  Los cálculos pueden contener errores debido a errores en los datos obtenidos, como errores de reloj, de órbita, rebotes de la señal, etc.

  Existe una técnica que permite conocer la posición actual con más precisión, y es la que más se utiliza en robótica, se conoce como GPS diferencial y consiste en corregir la desviación del robot en función de la desviación recibida en una posición conocida (Figura 1.6(b)).

  Este tipo de técnicas suelen utilizarse sobre todo en exteriores para localizar al robot con precisión, como por ejemplo en la competición *Grand Challenge*, donde varios vehículos autónomos compiten por llegar desde un punto de estados unidos hasta otro en el menor tiempo posible y sin intervención humana.



(a) (b)

Figura 1.6: Error en la estimación de la posición usando odometría (a). Funcionamiento GPS diferencial (b).

- Otros sensores: Utilizando otros sensores, como las cámaras, puede determinarse la posición del robot en un mundo conocido analizando la información obtenida:

  - Balizamiento: Se establecen una serie de balizas que pueden ser percibidas por el robot en posiciones previamente conocidas, éstas pueden ser tanto objetos con unas determinadas características como marcas realizadas sobre paredes o suelos, aunque no tienen porqué ser necesariamente visuales, ya que también pueden utilizarse otros sensores como el sensor láser.

    Una vez que se han localizado una o varias balizas se puede determinar la posición del robot respecto a las balizas, ya sea por triangulación o por trilateración:

    - Triangulación: Cálculo de la posición utilizando el ángulo en el que encuentras las balizas.
    - Trilateración: Cálculo de la posición utilizando la distancia que existe hasta las balizas. Este tipo de cálculo es difícil de realizar utilizando cámaras, y es más común cuando se utilizan sensores de ultrasonido o sistemas GPS.

    Este tipo de técnicas han sido utilizadas en la RoboCup para facilitar la localización de los robots dentro del campo, para lo cual se utilizaban una serie de cilindros en distintos puntos conocidos con un código de colores característico (Figura 1.7). Estas balizas se han eliminado recientemente y ahora sólo pueden utilizarse para la localización elementos naturales del campo, como las líneas o las porterías.

(a)

Figura 1.7: Localización con balizas en la RoboCup.

- Localización probabilística: Se determina la posición del robot por probabilidad, haciendo que cada posible posición en el mundo tenga una probabilidad entre 0 y 1 que indique si el robot puede encontrarse en esa posición, utilizando para ello la información recibida con los sensores.

  Esto se realiza utilizando un mapa de probabilidad en el que calculamos la probabilidad de cada observación instantánea, y donde se acumula la probabilidad de la última observación con las previamente calculadas mediante fusión de Bayes.

  Esta técnica debe permitirnos tanto localizar al robot en una posición absoluta dentro del mundo en el que se encuentra, como recuperarnos de posibles errores cuando el robot es movido a otra posición (lo que se conoce como problema del secuestro).

  Algunos de los algoritmos más utilizados hasta el momento basados en este tipo de localización son los filtros de partículas o el método de Montecarlo.

  La localización probabilística será utilizada en nuestro proyecto y la detallaremos en profundidad en el capítulo 5.

También se han realizado diversos proyectos en el grupo de Robótica relacionados con la localización, como [López Fernández, 2005], que desarrolló un algoritmo de localización con visión para el robot Pioneer utilizando el método de Montecarlo, o [Kachach, 2005], cuyo proyecto utilizaba para la localización en 2D un sensor láser en el robot Pioneer.

Capítulo 2

Objetivos

Una vez presentado el contexto 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 principal objetivo del proyecto es la localización de un robot dentro del terreno de juego de la RoboCup mediante la percepción espacial de las porterías, utilizando para ello exclusivamente la visión artificial y comprobando experimentalmente nuestros resultados en simuladores. Las simulaciones se realizarán utilizando el robot Nao de Aldebaran Robotics, pero no es un objetivo de este proyecto el utilizar el robot real ni imágenes reales.

Además, el proyecto contará con varios subobjetivos que nos permitirán alcanzar el objetivo principal y que se describen a continuación:

1. A partir de las imágenes recibidas desde las cámaras, tendremos que detectar las porterías del campo teniendo en cuenta el reglamente de la RoboCup, lo que nos servirá para tener puntos de referencia para la localización. Además los algoritmos deberán ser capaces de detectar posibles elementos externos que nos dificulten la detección, como la pelota, otros jugadores o los objetos que se encuentran fuera del campo, siempre que no obstaculicen la visión de la portería.

2. Tras detectar la portería en la imagen, deberemos ser capaces de calcular la posición del robot mediante localización 3D instantánea, es decir, a partir de una única imagen

con la suficiente información tendremos que ser capaces de situar al robot dentro del campo.

La localización del robot sólo deberá realizarse cuando se vea alguna de las dos portería del campo de forma completa, no siendo un objetivo de este proyecto calcular la localización cuando se vea la portería de forma incompleta.

3. Una vez que consigamos localizarnos a partir de una única imagen, el siguiente subobjetivo será poder calcular la posición del robot a partir de varias observaciones tomadas a lo largo del tiempo, utilizando para ello la acumulación de observaciones mediante probabilidad y haciendo uso de filtros de Bayes.

4. Los algoritmos desarrollados deberán ser validados experimentalmente sobre distintas plataformas de simulación, lo que nos permitirá mejorar los algoritmos sin necesidad de usar el robot real, y además nos asegurará que el funcionamiento está suficientemente depurado sin depender de características específicas de cada plataforma subyacente.

5. La aplicación desarrollada deberá tener una interfaz gráfica de usuario que facilite la selección de los distintos algoritmos y ayude en la muestra de resultados. Con el objetivo de facilitar la depuración, se deberá representar dentro de la aplicación un campo simulado en 3 dimensiones, en el que se mostrará la localización actual del robot, así como la probabilidad acumulada en el caso de utilizar los algoritmos probabilísticos.

Además de lo anterior, tendremos que tener en cuenta que el robot cuenta con 6 grados de libertad, 3 grados para su posición y otros 3 para la inclinación de la cámara, por lo que la estimación de la posición será más compleja que en escenarios con menos grados de libertad. Esto supone un salto en la complejidad respecto a otros trabajos realizados dentro del grupo de robótica, ya que hasta ahora sólo se habían realizado proyectos que calculaban la posición del robot en 2D.

2.2. Requisitos

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

- Plataforma: Nuestro proyecto deberá hacer uso de la arquitectura de desarrollo JdeRobot, con la que trabaja el grupo de robótica de la URJC. Esta arquitectura está desarrollada en los lenguajes de programación C, C++ y Python, sobre el sistema operativo Linux, por lo tanto se establece como requisito de partida el desarrollo del proyecto en alguno de estos lenguajes y en este sistema operativo.

- Visión artificial: Para la localización del robot, deberemos hacer uso únicamente de una cámara, no pudiendo utilizar otro tipo de sensores que nos ayuden en esta tarea, y obligando a hacer uso de la visión artificial.

- Tamaños de las imágenes: Las imágenes recibidas de las cámaras pueden tener diversas dimensiones, por lo que los algoritmos deberán funcionar independientemente del tamaño de las imágenes recibidas.

- Eficiencia de los algoritmos: Los algoritmos utilizados deberán ser lo suficientemente eficientes como para ser ejecutados de forma iterativa en el Robot Nao simulado, haciendo que se ejecute de forma ligera y sin consumir muchos recursos, para dejar el procesador libre para que lo utilicen otras funcionalidades. Además, al cumplir este requisito, conseguiremos que el código sea empotrable en el código del robot real, cuyas características hardware son muy limitadas y que se describirán en detalle en el capítulo 3.1.

- Precisión: La precisión conseguida en los distintos algoritmos que desarrollemos deberá ser del orden de centímetros, siendo aceptable un error en la localización en 3D de menos de 40 centímetros.

- Robustez: En cuanto a la robustez del algoritmo, no será necesario evitar oclusiones en la imagen, por ello, sólo necesitaremos detectar las porterías correctamente cuando obtengamos una imagen con la portería completa y sin elementos que nos obstaculicen.

- Licencia: El código del proyecto es software libre, por lo que será liberado bajo la licencia **GPLv3**[1].

[1] http://www.gnu.org/licenses/gpl-3.0-standalone.html

2.3. Metodología de desarrollo

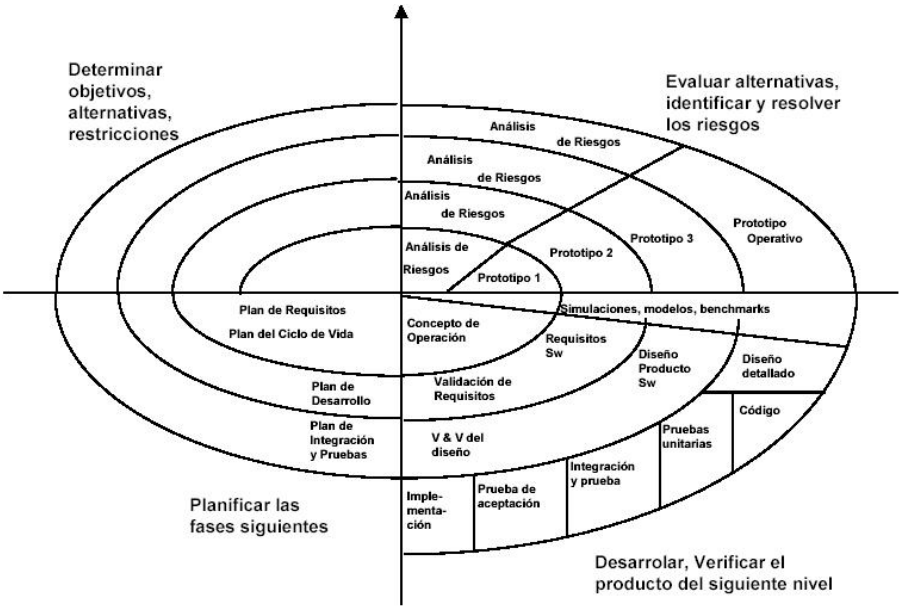
En el desarrollo de los componentes software de nuestro trabajo, el modelo de ciclo de vida utilizado ha sido el modelo en espiral basado en prototipos, ya que permite desarrollar el proyecto de forma incremental, aumentando la complejidad progresivamente y hace posible la generación de prototipos funcionales.

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 software. Dentro de cada ciclo del modelo en espiral se pueden diferenciar 4 partes principales que pueden verse en la figura 2.1, y donde cada una de las partes tiene un objetivo distinto:

- Determinar objetivos: Se establecen las necesidades que debe cumplir el producto 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 reducirlos 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.



(a)

Figura 2.1: Modelo en espiral.

2.3.1. Plan de trabajo

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 JdeRobot: Primero desarrollamos un componente de prueba para familiarizarnos con la plataforma de desarrollo software JdeRobot, ya que iba a ser la utilizada en el proyecto. Este primer componente, al que llamamos *Headtracking*, consistía en generar una escena en 3D en la que nos movíamos según el movimiento de nuestro propio cuerpo. Esta aplicación se basaba en el proyecto [Chung Lee, 2008] y puede verse detallada en el apéndice A.

   Además en este primer componente utilizamos algunas tecnologías que nos resultarían de utilidad en la aplicación final, como la creación de interfaces con GTK/Glade o la creación de escenas en 3D con OpenGL.

2. Introducción a la visión artificial: Para tomar un primer contacto con las técnicas visuales utilizadas en la visión artificial realizamos un nuevo componente para JdeRobot llamado *Opencvdemo*, que hacía uso de distintas funciones de la librería OpenCV para el tratamiento de imágenes. Este componente será descrito más adelante en el capítulo 3.6.

3. Detección de líneas: Aplicando lo aprendido en la fase anterior y buscando información en otros proyectos [Hidalgo Blázquez, 2006], [Smith *et al.*, 2006], desarrollamos un algoritmo para detectar líneas del campo. Para ello, creamos un campo de fútbol a pequeña escala que nos permitió realizar numerosas pruebas, utilizando cámaras para obtener las imágenes que eran analizadas por los algoritmos.

4. Detección de porterías en imagen: En esta fase generamos un simulador empotrado en nuestra aplicación que simulaba una portería y nos permitía movernos alrededor de ella. Con este simulador realizamos varios algoritmos que nos permitiesen detectar las esquinas de la portería usando la visión artificial.

5. Localización 3D instantánea: Una vez que eramos capaces de detectar las porterías en una imagen, procedimos a desarrollar los algoritmos para la localización 3D instantánea que describiremos más adelante en el capítulo 4. En estos algoritmos, se utilizó la librería *Progeo* de JdeRobot, que nos facilitó los cálculos geométricos para obtener la posición del robot, además de la librería GSL para realizar cálculos matriciales, utilizada ya en otros proyectos como [Kachach, 2008].

6. Comunicación con el robot Nao: Para la utilización de los simuladores de forma definitiva, realizamos un *driver* para JdeRobot, al que llamamos *Naobody*, para mejorar la comunicación con el robot Nao, y que nos permitió obtener las imágenes directamente de las cámaras simuladas del Nao, así como mover sus articulaciones para moverle por el campo a voluntad.

7. Localización 3D probabilística: Tras estudiar varias técnicas probabilísticas ya existentes [López Fernández, 2005], [Cañas and Matellán, 2002], extendimos los algoritmos de localización instantánea para conseguir una localización probabilística más robusta utilizando filtros de Bayes, como se verá en el capítulo 5. Además, utilizamos los algoritmos previamente desarrollados para la detección de líneas, para conseguir más fuentes de información con las que conocer la posición del robot.

En mi ficha web personal del grupo de robótica de la URJC [2] pueden encontrarse imágenes y vídeos que muestran las distintas etapas anteriormente descritas.

[2] http://jde.gsyc.es/index.php/User:Eperdes

Capítulo 3

Entorno y plataforma de desarrollo

En este capítulo vamos a describir los elementos empleados en el desarrollo del proyecto, tanto hardware como software. Además describiremos varias aplicaciones que hemos desarrollado y que nos han sido de ayuda para alcanzar los objetivos del proyecto.

El sistema operativo utilizado para el desarrollo software ha sido Ubuntu 8.04, una de las distribuciones de GNU/Linux más importantes actualmente, y que está basada en Debian. Este sistema operativo es de libre distribución y soporta oficialmente las arquitecturas hardware Intel x86 y AMD64, que se corresponden con las arquitecturas de los ordenadores que hemos utilizados.

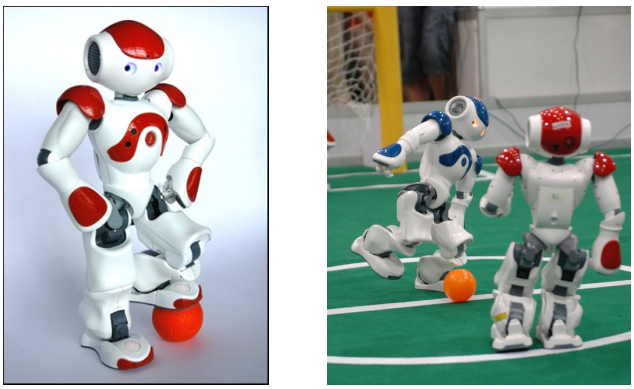
3.1. Robot Nao

El robot Nao 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. Se trata de un robot humanoide desarrollado por Aldebaran Robotics que actualmente se encuentra en su versión Nao RoboCup Edition V3, lanzada en enero de 2009. Podemos ver algunas imágenes de este robot en la figura 3.1.

Los algoritmos desarrollados en nuestro proyecto utilizan para su simulación a este robot, por lo que a pesar de no ser utilizado en la realidad, debemos tener en cuenta sus características, ya que condicionan la solución final. Algunas de sus características principales son:

- 21 grados de libertad, permitiendo una gran amplitud de movimientos (figura 3.2(a)).
- Detectores de presión en pies y manos, conocidos como FSR.

- 2 cámaras con distintas zonas de visión (figura 3.2(b)).
- 4 sensores de ultrasonido.
- Inerciales, de gran utilizad en caso de caídas.
- Ordenador incorporado con comunicación vía Ethernet o Wi-Fi, permitiendo así las comunicaciones inalámbricas.
- LEDs en diversas partes del cuerpo, de gran ayuda para conocer el estado del robot.



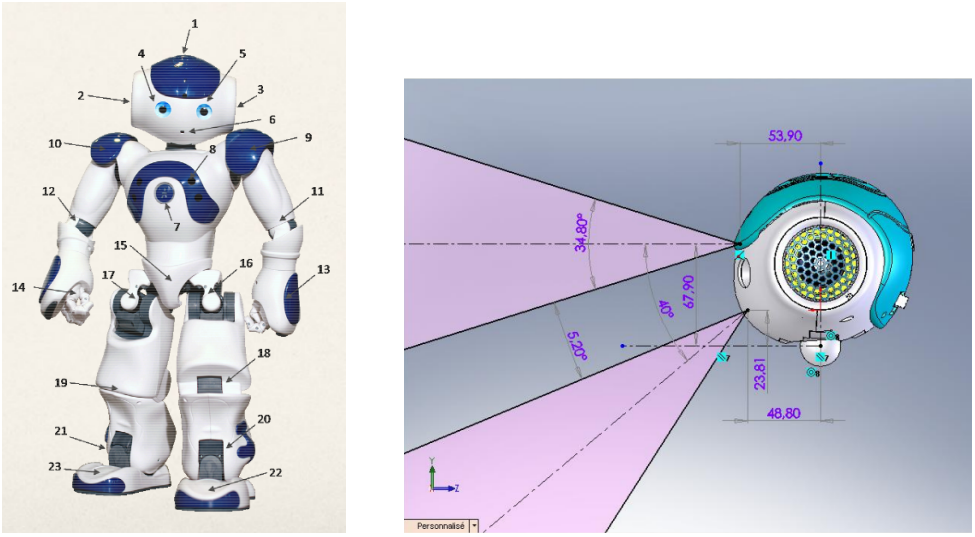
(a) (b)

Figura 3.1: Robot Nao de Aldebaran Robotics (a). Imagen de la RoboCup 2008 (b).

Además de las características anteriores, el robot también dispone de un sistema multimedia (compuesto por 4 micrófonos y 2 altavoces hi-fi) que le permite comunicarse mediante voz, un localizador de sonido y algoritmos de reconocimiento facial, aunque estas características no son utilizadas por el momento en el desarrollo de software para la RoboCup.

Como procesador utiliza una CPU modelo x86 AMD GEODE 500MHz y usa como sistema operativo Linux. Para el movimiento autónomo, usa una batería que le permite estar funcionando durante 45 minutos en espera o durante 15 minutos caminando.

El robot Nao dispone de una interfaz de programación que permite interactuar con él de una forma intuitiva, utilizando la arquitectura software Naoqi que veremos en la próxima sección.



(a) (b)

Figura 3.2: Grados de libertad del Robot Nao (a). Cámaras del Robot Nao (b).

3.1.1. Naoqi

Naoqi es una arquitectura software multiplataforma para la comunicación y el desarrollo de software del robot Nao, y que permite crear un entorno distribuido donde varios binarios pueden comunicarse entre sí. Soporta el desarrollo en los lenguajes de programación C++, Python y Urbi, siendo además compatible con la plataforma de simulación Webots.

Esta arquitectura software nos ha sido de gran ayuda en la comunicación con el robot simulado, aunque también es posible usarlo con el robot Nao real.

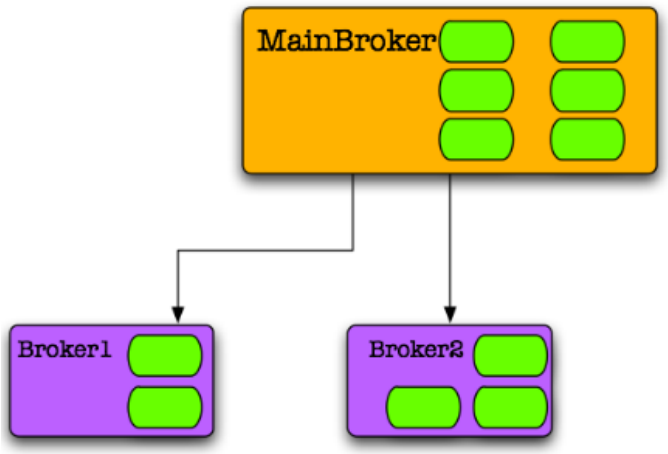
Naoqi se compone de tres componentes básicos, los *brokers*, los módulos y los *proxies*:

- Brokers: Son ejecutables que se conectan a un puerto e IP determinados, todos los *brokers* están conectados formando un árbol, cuya raíz es un *broker* principal (MainBroker), como puede verse en la figura 3.3.
- Módulos: Son clases que derivan de ALModule y que presentan una funcionalidad concreta, como por ejemplo ALMotion para el acceso a los motores y NaoCam para el acceso a la cámara del robot. Estos módulos pueden ser cargados por un *broker*.
- Proxies: Sirven para crear un representante de un módulo a través del cual utilizamos las operaciones que nos sirve ese módulo.

Algunos de los módulos principales que hemos utilizado son:

- ALMotion: Es el módulo encargado de las funciones de locomoción, se utiliza principalmente para establecer los ángulos de apertura de las distintas articulaciones. En general, los parámetros que acepta para cada articulación son el grado final en el que quedará la articulación, la velocidad de la transición, o bien, la duración que tendrá la transición. También es el módulo encargado de controlar el centro de masas, además de permitirnos llamar a funciones que directamente hacen al robot andar o girar.

- NaoCam: Es el módulo encargado de comunicarse con la cámara superior del robot Nao. Mediante este módulo obtenemos las imágenes de la cámara en el formato y dimensiones que pongamos como parámetros, permitiendo así conseguir la imagen en el formato necesario para procesarla después.



(a)

(b)

Figura 3.3: Árbol de *brokers* (a). Módulos del *broker* principal (b).

3.2. JdeRobot

El proyecto se ha desarrollado en la plataforma JdeRobot [1]. Esta plataforma ha sido creada por el grupo de robótica de la URJC, se trata de una plataforma de desarrollo de software para aplicaciones con robots y visión artificial. Se ha elegido JdeRobot por la facilidad con la que se pueden utilizar otras aplicaciones ya creadas para esta plataforma de desarrollo, además al haber sido utilizada en proyectos similares, han podido reutilizarse partes del código de otros proyectos.

JdeRobot está desarrollado en C, C++ y Python 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 específica que podrá 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 órdenes 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 generar este comportamiento se han desarrollado una serie de drivers, que actúan como *plugins* y son los encargados de dialogar con los dispositivos hardware concretos.

En particular, para la realización de nuestro proyecto, hemos utilizado algunos de los drivers que proporciona la plataforma, que son:

- Video4linux: Driver encargado de la obtención de imágenes desde las cámaras que realiza la actualización de la imagen obtenida de la cámara automáticamente, usado en diversas pruebas realizadas con cámaras reales.
- Imagefile: Que facilita la lectura de imágenes almacenadas en el disco duro, de gran ayuda al usar imágenes reales sacadas por el robot Nao real.
- Gazebo: Driver que nos permite mover y obtener imágenes de robots en el simulador Gazebo, el cual ha sido utilizado durante el desarrollo de nuestro trabajo.

[1] http://jde.gsyc.es/index.php/Main_Page

La versión utilizada de JdeRobot ha sido la 4.3.0, lanzada en abril de 2009, y que se compone de 17 esquemas y 12 drivers.

3.2.1. Progeo

Además de lo citado anteriormente, se ha utilizado la librería Progeo, proporcionada también por JdeRobot, una librería sobre geometría proyectiva que nos proporciona algunas funciones útiles para relacionar información visual de una cámara en 2D con información espacial en 3D. Esta relación se consigue principalmente mediante dos funciones:

- *project*: Función que nos permite obtener el píxel en 2 dimensiones del plano imagen de la cámara en el que proyecta un punto del espacio en 3 dimensiones.
- *backproject*: Esta función realiza la operación inversa, a partir de un píxel en 2 dimensiones en el plano imagen, permite obtener la recta proyectiva 3D del conjunto de puntos en 3 dimensiones que proyectan en ese píxel de la imagen.

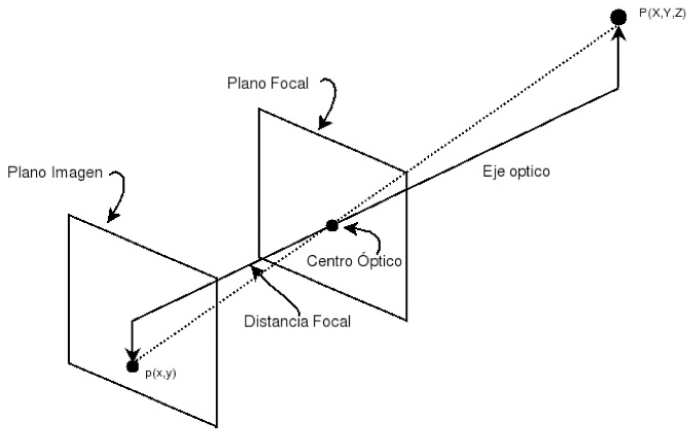
Antes de utilizar estas dos funciones, es necesario calibrar la cámara que estemos utilizando para que los cálculos hechos por la librería sean los correctos.

Para ello, es necesario saber que Progeo se basa en un modelo de cámara Pinhole (Figura 3.4), que se define por un conjunto de parámetros intrínsecos y extrínsecos. Los parámetros extrínsecos consisten en la posición de la cámara en 3 dimensiones, el foco de atención de la cámara y el roll, mientras que los intrínsecos son la distancia focal y la posición del píxel central.

Los valores extrínsecos de la cámara pueden establecerse por defecto, sabiendo que la posición en 3D de la cámara será siempre $(0, 0, 0)$. Los valores intrínsecos de las cámaras reales los hemos calculado utilizando el esquema de JdeRobot *Calibrador*, desarrollado en el proyecto [Kachach, 2008].

En el caso de las cámaras simuladas, los valores se han obtenido teniendo en cuenta cómo funcionan las cámaras en OpenGL, donde los parámetros intrínsecos que se utilizan son el ángulo horizontal y la relación de aspecto.

Así, la posición del píxel central, se obtendrá siempre dividiendo por dos el ancho y la altura de la imagen, ya que en OpenGL se utiliza un modelo de cámara ideal. Mientras que la distancia focal se obtendrá con la ecuación 3.1, donde f es la distancia focal que buscamos, *size* es el ancho o la altura de la imagen, dependiendo del simulador, y θ es el ángulo horizontal en radianes que utilice el simulador.



(a)

Figura 3.4: Modelo de cámara Pinhole.

$$f = \frac{size}{2 * tan\frac{\theta}{2}} \tag{3.1}$$

Progeo ha sido muy útil en el desarrollo del proyecto, puesto que al detectar determinados píxeles en las imágenes, ha sido necesario transformar sus coordenadas en 2D a 3D, utilizando para ello la función *backproject* de esta librería.

3.3. Webots y Gazebo

Para probar los desarrollos realizados para el robot Nao sin necesidad de utilizar el robot real, hemos utilizado dos simuladores, Webots[2] y Gazebo[3]. Se han utilizado dos simuladores mejor que uno solo para probar la robustez de los algoritmos, que funcionan sin cambios en ambas plataformas, y comprobar que estaban suficientemente depurados.

El primero de los simuladores, Webots, es un entorno de desarrollo creado por la compañía Ciberbotics utilizado para modelar, programar y simular robots móviles.

Con Webots se pueden configurar una serie de robots y objetos que interactúan entre sí en un entorno compartido. Además, para cada objeto se pueden establecer sus propiedades, tales como, forma, color, textura, masa, etc. También hay disponibles un

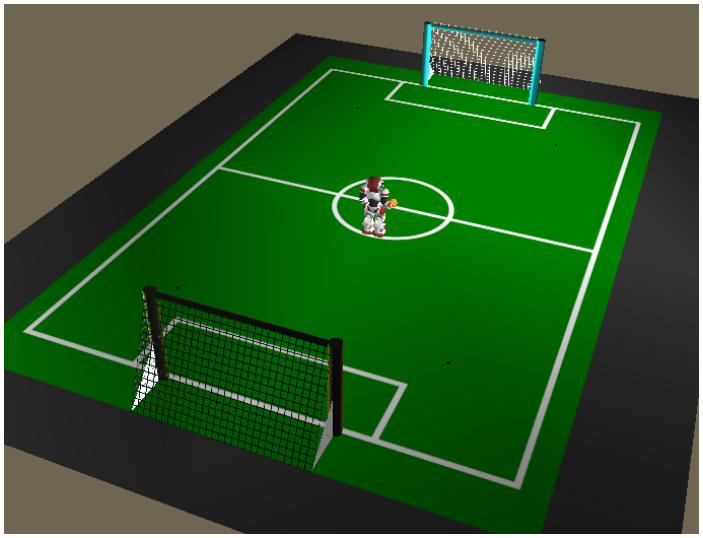
[2]http://www.cyberbotics.com/

[3]http://playerstage.sourceforge.net/gazebo/gazebo.html

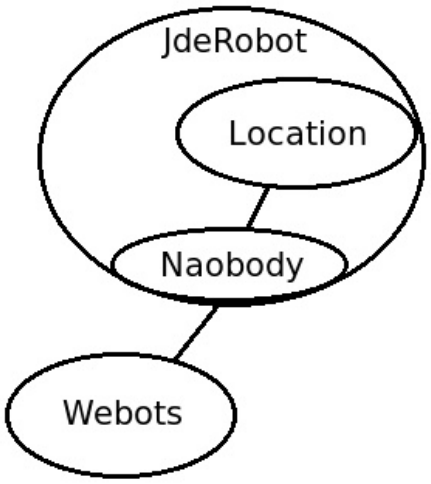
amplio conjunto de actuadores y sensores para cada robot, de esta manera podemos probar el comportamiento de la física del robot dentro de un mundo realista.

Dentro de la plataforma de desarrollo existen varios modelos de robots y objetos preestablecidos, entre los que se encuentra el robot Nao y los objetos necesarios para simular el campo de la RoboCup (figura 3.5(a)).

La comunicación con el robot Nao en Webots, se ha realizado mediante un driver para JdeRobot desarrollado por nosotros mismos que se explicará en la sección 3.7, y que se ejecuta de forma paralela a nuestro esquema principal (figura 3.5(b)).



(a)



(b)

Figura 3.5: Ejemplo de campo de la RoboCup con el simulador Webots (a). Comunicación con Webots en JdeRobot (b).

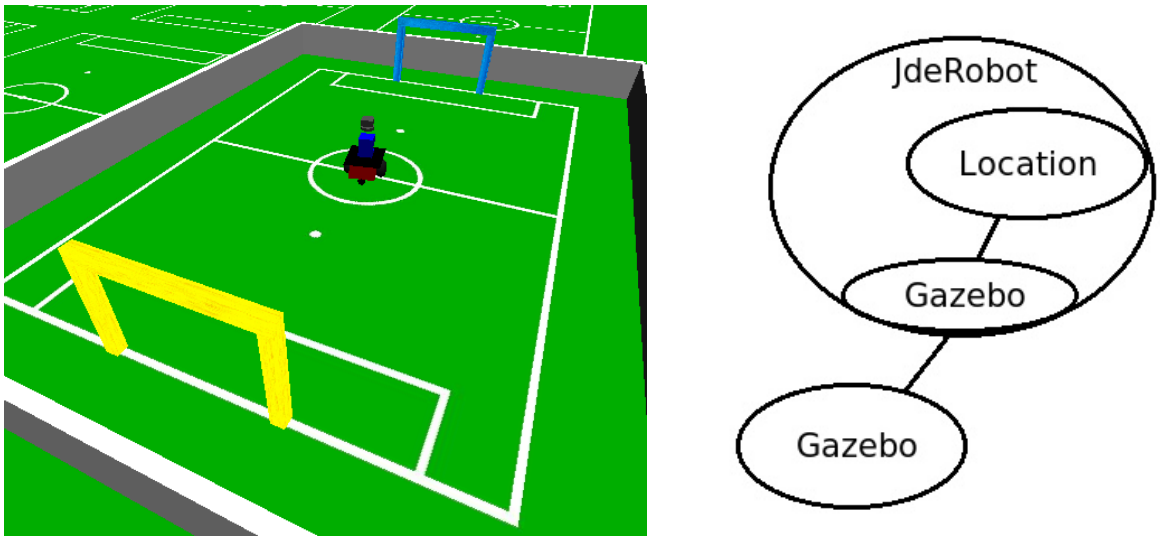
Por su parte Gazebo, es un simulador de múltiples robots en 3D, permitiendo simular también sensores y objetos con los que obtener realimentación e interactuar. Gazebo es software libre bajo licencia GPL y forma parte del proyecto Player/Stage. Proporciona varios modelos de robots, como el Pioneer2DX, Pioneer2AT y SegwayRMP, y diversos mundos en los que probar estos robots.

Además, al igual que Webots, permite configurar nuestros propios mundos, robots y objetos, en los que se pueden establecer las propiedades que creamos necesarias. Sin embargo, al contrario que en Webots, aún no existe el robot Nao ni el campo de fútbol de la RoboCup previamente diseñados.

Por ello, para realizar nuestras pruebas, hemos utilizado un mundo para Gazebo creado

por Francisco Rivas [4], en el que simula el campo de fútbol de la RoboCup según las reglas establecidas para la plataforma estándar en el año 2009 (figura 3.6(a)), y donde se utiliza como robot un Pioneer2DX, que si bien no es igual que el robot Nao, es válido para la simulación puesto que dispone de una cámara con la que ver el mundo y puede moverse a través de él.

Para comunicarnos con el robot en el simulador Gazebo se ha utilizado el driver Gazebo ya disponible en JdeRobot, y que nos ha permitido mover el robot, conocer su posición actual dentro del mundo y obtener las imágenes necesarias (figura 3.6(b)).



(a) (b)

Figura 3.6: Campo de la RoboCup generado por Francisco Rivas simulado con Gazebo (a). Comunicación con Gazebo en JdeRobot (b).

3.4. GTK+ y Glade

GTK+ es un conjunto de bibliotecas multiplataforma para crear interfaces gráficas de usuario en múltiples lenguajes de programación como C, C++, Java, Python, etc. GTK+ es software libre bajo licencia LGPL y es parte del proyecto GNU. Entre las bibliotecas que componen a GTK+, destaca GTK, que es la que realmente contiene contiene los objetos y funciones para la creación de la interfaz de usuario.

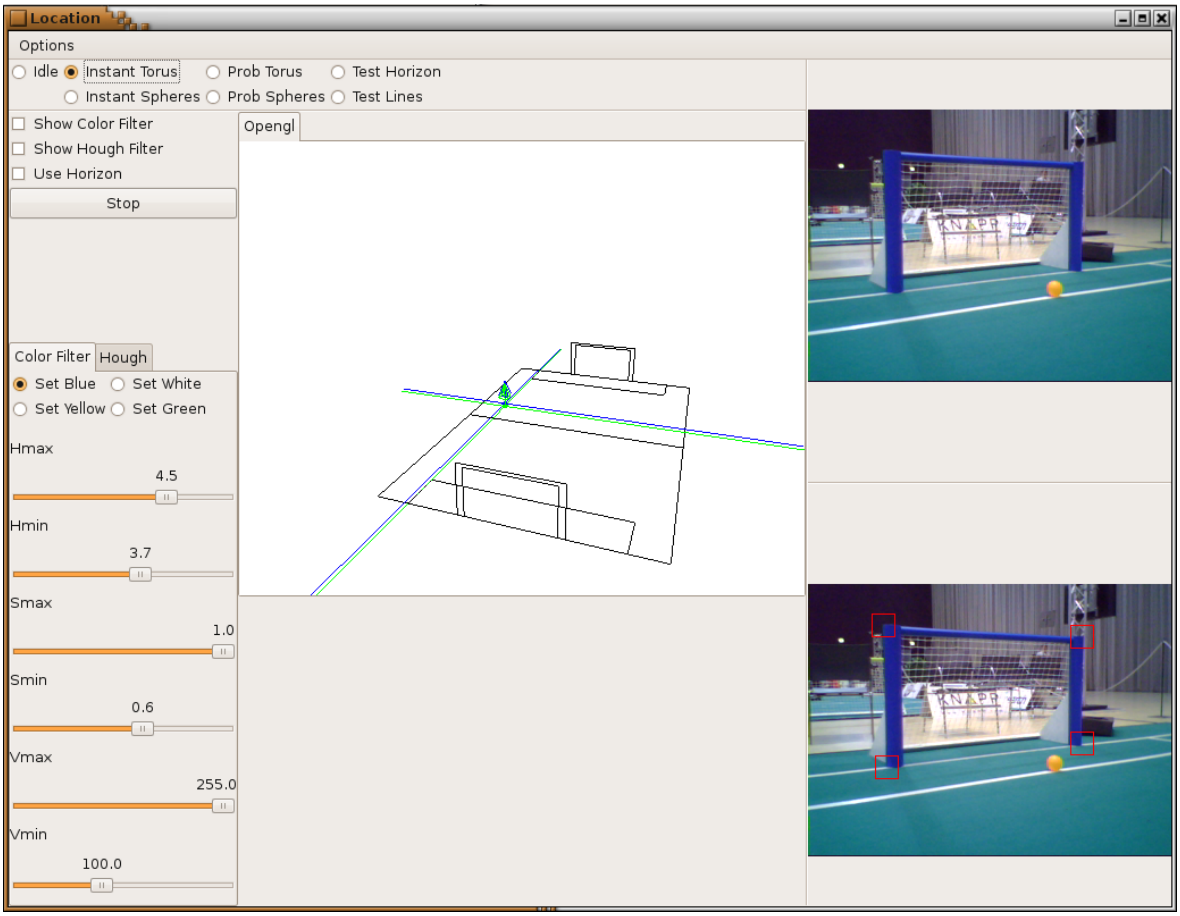
Son numerosas las aplicaciones desarrolladas con esta librería, algunas muy conocidas

[4]http://jde.gsyc.es/index.php/Frivas-pfc-itis

como el navegador web Firefox o el editor gráfico GIMP, por lo que puede comprobarse su gran potencia y estabilidad.

Para facilitar la creación de las interfaces, existe el diseñador de interfaces Glade, una herramienta de desarrollo visual de interfaces gráficas de GTK. Esta herramienta es independiente del lenguaje de programación, ya que genera un archivo en XML que debe ser leído e interpretado utilizando la librería Glade del lenguaje que se vaya a utilizar.

En nuestro proyecto, hemos utilizado GTK+ para la realización de las interfaces gráficas, ayudándonos del diseñador de interfaces Glade para simplificar el desarrollo, y cuyo resultado puede verse en 3.7.



(a)

Figura 3.7: Interfaz gráfica de nuestro proyecto.

3.5. OpenGL

OpenGL es una especificación estándar que define una API multiplataforma e independiente del lenguaje de programación para desarrollar aplicaciones con gráficos en 2D y 3D. A partir de primitivas geométricas simples, como puntos o rectas, permite generar escenas tridimensionales complejas. Actualmente es ampliamente utilizado en realidad virtual, desarrollo de videojuegos (figura 3.8) y en multitud de representaciones científicas.

Hemos utilizado esta API en nuestras aplicaciones para la representación en 3D del campo de la RoboCup, con el objetivo de mostrar la localización calculada por nuestros algoritmos y facilitar la depuración, como puede verse en la imagen de la sección anterior (3.7).

(a)

Figura 3.8: Captura del videojuego Counter Strike, desarrollado en OpenGL.

3.6. OpenCV

OpenCV es una librería de visión artificial desarrollada inicialmente por Intel. Está publicada sobre la licencia BSD, lo que permite su libre utilización en propósitos comerciales o de investigación.

Esta librería nos proporciona un extenso conjunto de funciones para trabajar con visión artificial cuyo desarrollo se ha realizado primando la eficiencia, para lo que se ha programado utilizando C y C++ optimizados, pudiendo además hacer uso del sistema de primitivas de rendimiento integradas en los procesadores Intel (IPP), que son un conjunto

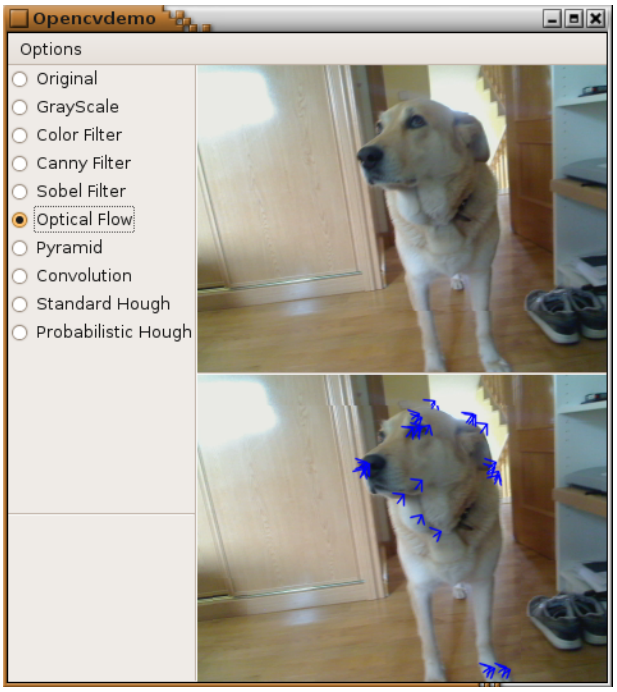
de rutinas de bajo nivel específicas en estos procesadores y que cuentan con una gran eficiencia.

Para tomar un primer contacto con el procesado de imágenes y la visión artificial, realizamos un esquema para JdeRobot que utilizase algunas funciones de la librería OpenCV, la cual también está disponible en el robot Nao, y que nos serviría para procesar las imágenes recibidas de las cámaras.

La finalidad del esquema era mostrar el potencial de la librería OpenCV, utilizando las funciones que incorpora para tareas de visión artificial, y así poder comprobar su alta eficiencia.

En el esquema hemos implementado las funcionalidades que pueden verse en la figura 3.9: Imagen en escala la grises, filtro de color HSV, filtro de bordes Canny, filtro de bordes Sobel, flujo óptico, pirámide multiresolución, matriz de convolución, transformada de Hough estándar y transformada de Hough probabilística.

Dentro de estas funcionalidades, algunas de ellas han sido utilizadas finalmente en la aplicación final, por lo que vamos a detallar en las siguientes secciones las más importantes.



(a)

Figura 3.9: Flujo óptico con Opencvdemo.

3.6.1. Filtro de color

Al obtener la imagen de la cámara, cada píxel se encuentra en formato RGB, es decir, cada píxel ocupa 3 bytes, el primero para la componente roja (Red), el segundo para la verde (Green) y el tercero para la azul (Blue). Debido a que este tipo de formato es poco robusto antes los cambios de luminosidad, decidimos convertir la imagen a formato HSV utilizando para ello la función de la librería OpenCV:

```
cvCvtColor( const CvArr* src, CvArr* dst, int code );
```

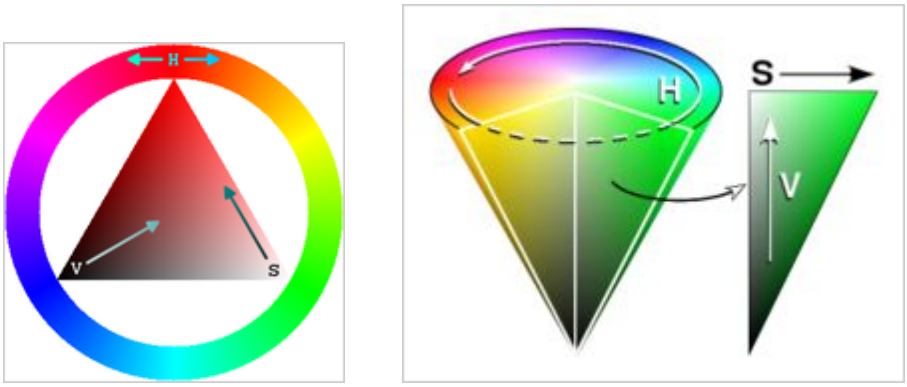
donde *src* es la imagen origen, *dst* es la imagen destino y *code* es el tipo de transformación, que en este caso es 'CV_RGB2GRAY' para pasar de RGB a escala de grises. Con esto, ahora los píxeles de la imagen están definidos por los componentes de tonalidad (Hue), saturación (Saturation) y brillo (Brightness):

- Tonalidad: La tonalidad es el tipo de color, se representa mediante un grado de ángulo cuyos valores van desde 0 hasta 360 grados. Cada grado corresponde a un color distintos, donde el 0 es el rojo, 120 el verde y 240 el azul.
- Saturación: Se representa como la distancia al eje de brillo negro-blanco, sus valores pueden ir desde el 0 al 100 %, cuanto menor sea la saturación de un color, existirá una mayor tonalidad grisácea.
- Brillo: Representa la altura en el eje blanco-negro, los valores al igual que en la saturación varían entre el 0 y 100 %, donde el 0 corresponde al negro y dependiendo de la saturación, 100 podría ser el blanco a un color más o menos saturado.

El formato HSV puede representarse visualmente con forma de cono o de rueda, como puede verse en la figura 3.10.

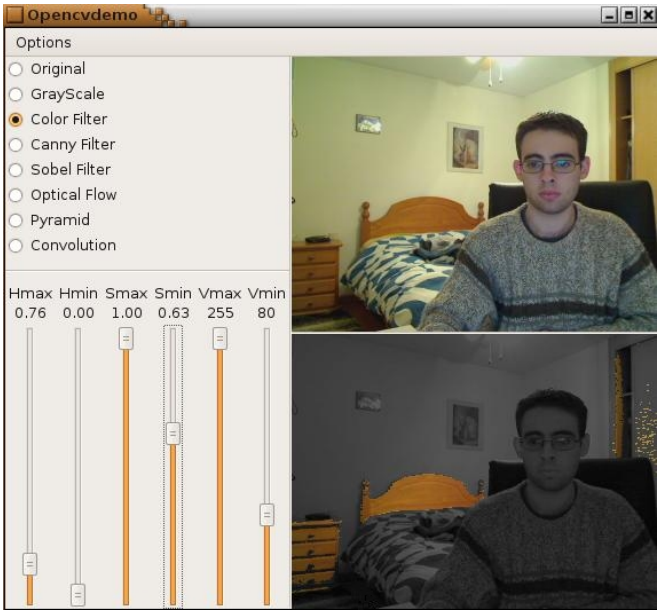
Una vez obtenida la imagen en formato HSV, recorremos toda la imagen píxel a píxel comprobando si sus valores H, S y V están entre un rango de valores para cada caso, los que cumplan esta condición los representamos en la imagen de salida con su color original, mientras que los que no lo hagan, se representarán en escala de grises, como puede observarse en la figura 3.11.

El filtro de color ha sido uno de los elementos más utilizados durante en el desarrollo, ya que con él hemos sido capaces de seleccionar objetos por su color, como las porterías o las líneas del campo.



(a) (b)

Figura 3.10: Representación HSV como una rueda (a) y como un cono (b).



(a)

Figura 3.11: Filtro de color HSV con Opencvdemo.

3.6.2. Filtro Canny

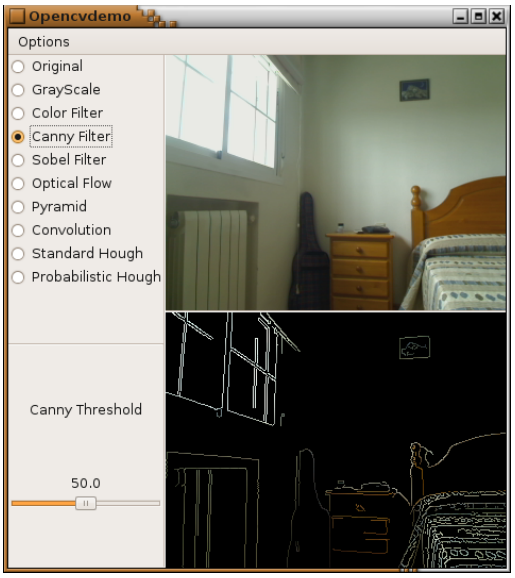
El filtro Canny es un algoritmo que detecta los bordes que existen en una imagen, siendo considerado como uno de los mejores métodos de detección de bordes. Utilizando matrices de convolución y derivadas detecta los puntos donde se produce un cambio brusco en el nivel de gris.

En OpenCV existe una función que realiza este algoritmo:

```
void cvCanny( const CvArr* image, CvArr* edges, double threshold1,
double threshold2, int aperture_size=3 );
```

donde *image* es la imagen de entrada, *edges* la imagen destino, *threshold1* y *threshold2* son umbrales utilizados por el algoritmo y *aperture_size* es un parámetro de apertura que por defecto es 3. La imagen de salida obtenida sera una máscara donde los píxeles que pasen el filtro Canny tendrán valor verdadero y el resto tendrán el valor falso. A partir de esta máscara, podemos generar una imagen donde los píxeles que pasaron el filtro Canny guarden su color original, mientras que el resto se muestren en color negro. De esta manera podremos obtener una imagen como la mostrada en 3.12.

Hemos usado este filtro en el proceso de detección de líneas y porterías, resaltando los bordes en la imagen con el fin de identificar los objetos que la componen.



(a)

Figura 3.12: Filtro de bordes Canny con Opencvdemo.

3.6.3. Transformada de Hough

La transformada de Hough es un algoritmo empleado para encontrar ciertas formas dentro de una imagen, como líneas o círculos.

El proceso realizado para conseguir encontrar cada línea consiste en recorrer cada punto que se desea averiguar si es parte de una línea, es decir, cada punto que se ha identificado anteriormente como borde, calculando las posibles líneas que puedan formar parte de ese punto. Al realizarse esto para todos los puntos, se determina qué líneas fueron las que más puntos posibles obtuvieron y se toman éstas como líneas en la imagen.

En nuestro caso, hemos utilizado este algoritmo para el reconocimiento de líneas mediante la función de OpenCV:

```
CvSeq* cvHoughLines2( CvArr* image, void* line_storage, int method,
                      double rho, double theta, int threshold,
                      double param1=0, double param2=0 );
```

donde *image* es la imagen de entrada, *line_storage* es un almacenamiento de memoria para los cálculos propios del algoritmo, *method* es el tipo de algoritmo que se quiere utilizar, *rho* es la distancia entre píxeles que están relacionados, *theta* el ángulo de resolución en radianes, *threshold* es un umbral utilizado por el algoritmo y *param1* y *param2* son parámetros que tienen un significado distinto dependiendo del método que se haya seleccionado.

Existen tres métodos distintos que se pueden utilizar, 'CV_HOUGH_STANDARD', 'CV_HOUGH_PROBABILISTIC' y 'CV_HOUGH_MULTI_SCALE', en nuestro esquema sólo hemos utilizado los dos primeros, puesto que el tercero no ofrecía diferencias significativas.

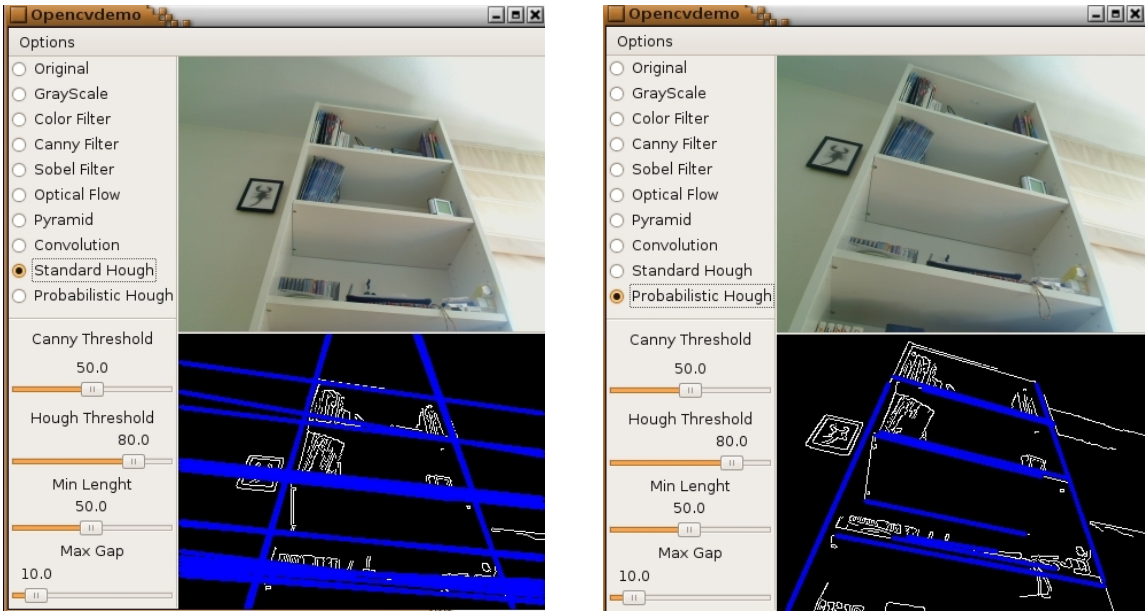
Utilizando Hough estándar, los parámetros *param1* y *param2* no se utilizan, por lo que pueden tomar cualquier valor sin afectar al resultado final, como consecuencia obtendremos un conjunto de líneas propagadas hasta el infinito.

Por el contrario, si utilizados Hough probabilístico, *param1* será la distancia mínima que debe tener una línea para considerarla válida y *param2* es el hueco que puede haber entre 2 píxeles para considerar que pertenecen a la misma recta. En este caso obtendremos un conjunto de líneas en las que se nos marca su principio y final.

La diferencia entre estos dos algoritmos puede apreciarse mejor en la figura 3.13.

En nuestro desarrollo, hemos decidido utilizar la transformada de Hough probabilística respecto a la estándar, puesto que nos era de gran ayuda conocer dónde comenzaba y terminaba cada línea.

Esta función ha sido necesaria a la hora de buscar las líneas y los extremos de la portería, puesto que la propiedad principal de ambos objetos es que están formados por líneas rectas.



(a) (b)

Figura 3.13: Aplicación de la transformada de Hough estándar (a) y probabilística (b).

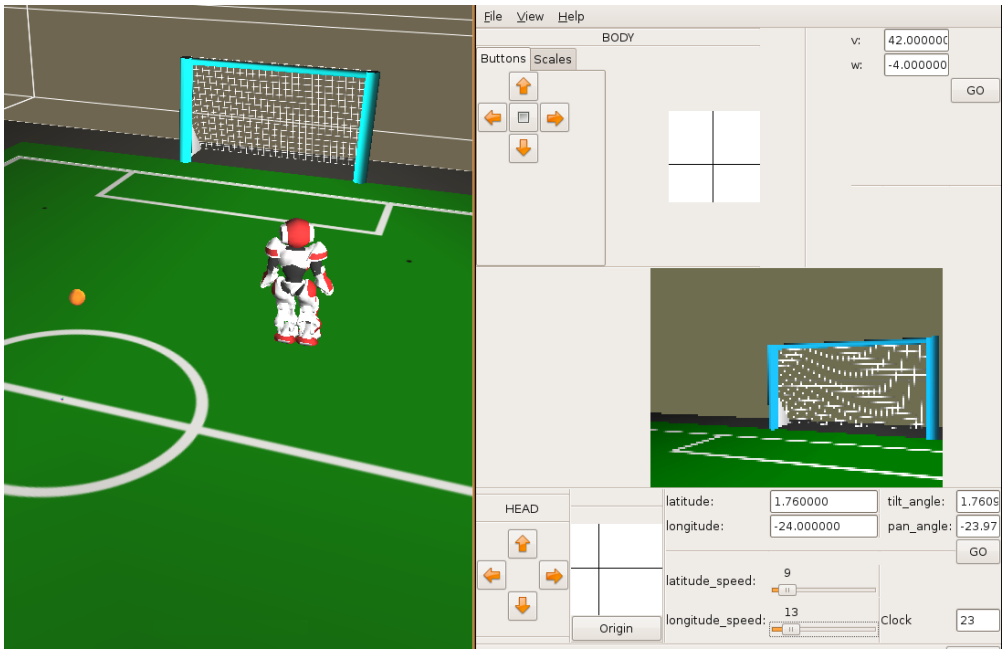
3.7. Naobody

Con el propósito de facilitar la comunicación con el robot Nao, ya sea simulado o real, hemos creado un driver para JdeRobot junto con Francisco Rivas [5], al que hemos llamado Naobody. Este driver, desarrollado en C y C++, utiliza la arquitectura de desarrollo Naoqi para la comunicación con el robot Nao.

Actualmente el driver permite obtener imágenes de la cámara del Nao de forma automática, mover la cabeza del robot y hacer que el robot camine tanto frontal como lateralmente. Para conseguir lo anterior, se ha generado una API que aumenta el nivel de abstracción respecto al proporcionado por Naoqi.

Al utilizar el driver desde alguno de los esquemas de JdeRobot, sólo es necesario incrementar o decrementar una serie de variables como si se tratasen de variables locales, y el driver se encarga automáticamente de llamar a las funciones necesarias para comunicarse con el robot.

Para probar este driver, hemos utilizado un esquema desarrollado por Francisco Rivas, llamado *NaoOperator*, que hace uso del driver Naobody y nos permite mover el robot por el mundo de una forma simple (figura 3.14).



(a)

Figura 3.14: Esquema *NaoOperator* utilizando el driver Naobody y el simulador Webots.

[5] http://jde.gsyc.es/index.php/Frivas-pfc-itis

Capítulo 4

Localización 3D instantánea

El primer tipo de localización que vamos a estudiar es la instantánea, donde a partir de una sola imagen, somos capaces de obtener la posición absoluta del robot.

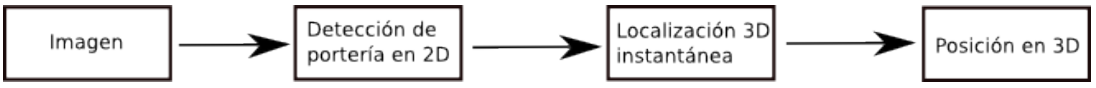
En este capítulo, vamos a realizar primero una descripción del diseño general utilizado, para después describir con detalle y evaluar experimentalmente cada uno de los algoritmos desarrollados.

4.1. Diseño general

Para lograr la detección 3D instantánea, hemos seguido el proceso que se muestra en la figura 4.1, donde a partir de una imagen recibida desde las cámaras, reales o simuladas, obtenemos como resultado una posición en 3D donde estará situado nuestro robot.

El primer paso a realizar, es analizar la imagen recibida en busca de alguna de las dos porterías del campo, para obtener los píxeles de la imagen en los que se encuentran sus 4 esquinas y conocer si estamos viendo la portería azul o la amarilla.

Estos datos calculados al detectar la portería, son la entrada a los algoritmos de localización desarrollados, que a partir de una serie de cálculos geométricos son capaces de determinar la posición en 3D (x, y, z) en la que se encuentra el robot.



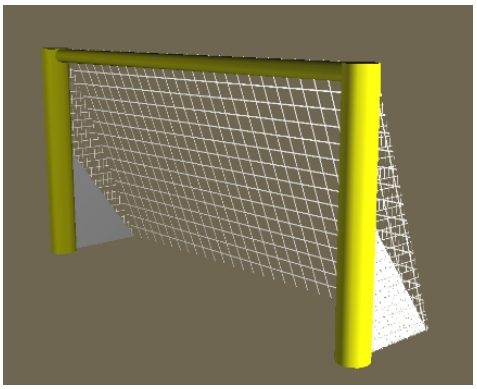
(a)

Figura 4.1: Proceso seguido en la localización instantánea.

El programa desarrollado cuenta con dos hilos de ejecución, el primero de ellos es el encargado de realizar los cálculos necesarios para obtener la localización, mientras que el segundo se encarga de mostrar en la interfaz de usuario los resultados obtenidos de forma gráfica, además de permitir configurar distintos parámetros a utilizar en los algoritmos.

4.2. Detección de la portería en la imagen

Las porterías son uno de los elementos principales que puede utilizar el robot para estimar su posición dentro del campo de la RoboCup. En las reglas para la plataforma estándar de la RoboCup en el año 2009 [RoboCup, 2009], se establece que las porterías utilizadas serán una de color amarillo y otra de color azul, teniendo un aspecto similar al de la imagen 4.2 y donde las dimensiones de ambas porterías serán las que se indican en las figuras 4.3(a) y 4.3(b).

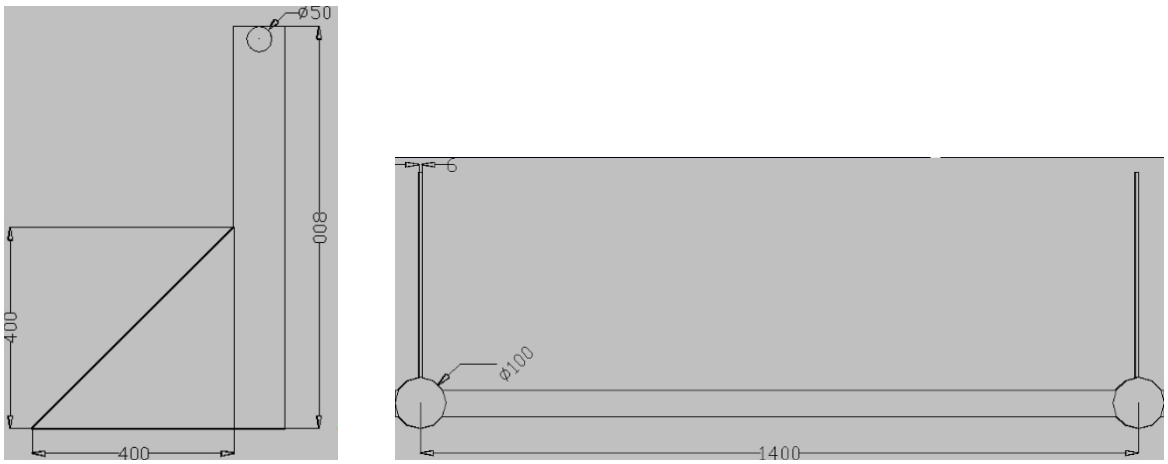


(a)

Figura 4.2: Aspecto simulado de la portería.

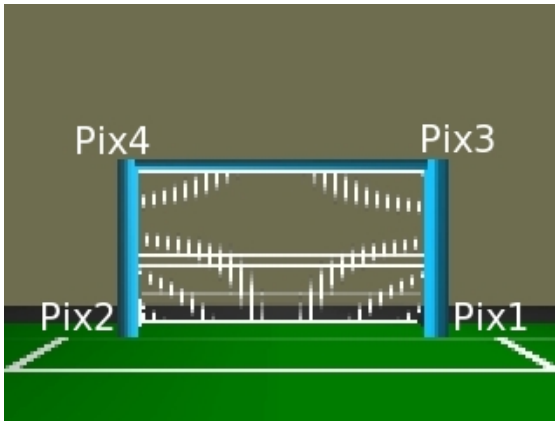
El objetivo de nuestro algoritmo es detectar alguna de las dos porterías, localizando las 4 esquinas indicadas en la figura 4.4 como *Pix1*, *Pix2*, *Pix3* y *Pix4*. En el caso de no encontrar todas las esquinas de la portería, la detección no será válida y no se utilizará en los algoritmos de localización, como ya se especificó en los objetivos del capítulo 2.

El algoritmo de detección cuenta con una serie de etapas, en los que la salida de cada una de las etapas sirve como entrada a la siguiente. Así, los pasos a seguir son: filtro de color, filtro Canny, transformada de Hough y cálculo de esquinas.



(a) (b)

Figura 4.3: Dimensiones de la portería según el estándar 2009 de la RoboCup.



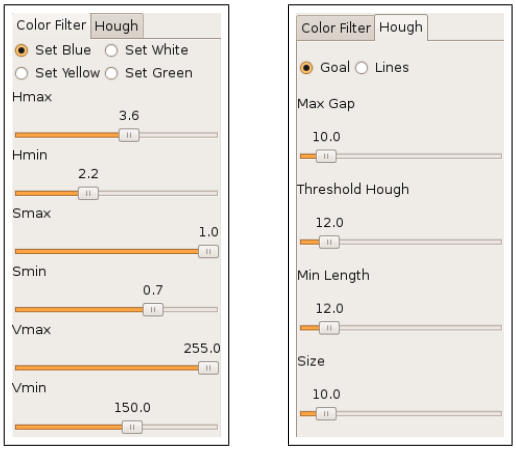
(a)

Figura 4.4: Esquinas de la portería.

Filtro de color

El primer paso del algoritmo consiste en realizar un filtro de color HSV como el explicado en el capítulo 3.6.1. Teniendo en cuenta que la portería que podemos encontrar en la imagen es tanto la azul como la amarilla, realizamos dos filtros de color simultáneos en la imagen, cada uno con uno de los colores, y guardamos el resultado en dos imágenes distintas.

Los tonos de azul y amarillo a buscar dependen de la plataforma que estemos utilizando (Webots, Gazebo o Nao real), e incluso pueden cambiar en la realidad por la iluminación o la luz solar. Por ello, hemos incluido dentro de nuestra aplicación una herramienta para poder seleccionar los valores HSV de los colores que queremos encontrar en la imagen. Esta herramienta nos permite modificar 6 parámetros, que se corresponden con los máximos y mínimos de cada uno de los componentes que forman el formato HSV (figura 4.5(a)).

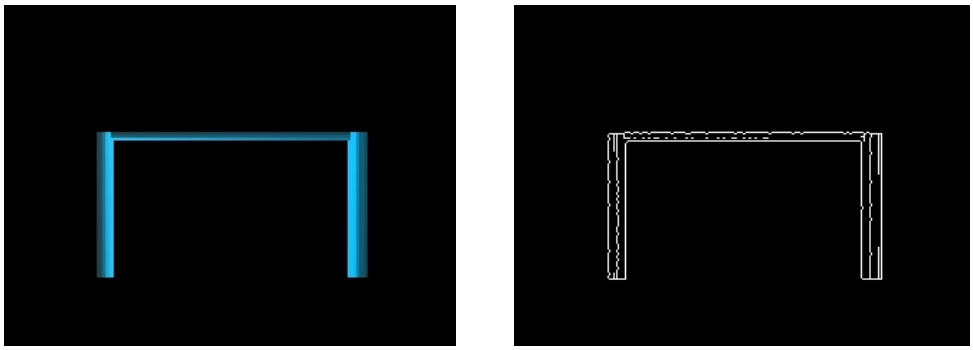


(a) (b)

Figura 4.5: Parámetros del filtro de color (a) y la transformada de Hough (b).

Filtro Canny

Una vez obtenidas las dos imágenes resultantes del filtro de color, nos quedamos con la imagen obtenida para el filtro de color azul, cuyo resultado puede ser similar al de la figura 4.6(a). Sobre esta imagen realizamos un filtro Canny para obtener los bordes de la portería, como se explicó en el capítulo 3.6.2, obteniendo la imagen que se muestra en 4.6(b).



(a) (b)

Figura 4.6: Filtro de color (a) y filtro Canny (b).

Transformada de Hough

Con la imagen resultante del filtro Canny, obtenemos las líneas rectas que hay en la imagen usando la transformada de Hough probabilística, explicada en 3.6.3, con lo que previsiblemente obtendremos líneas rectas que definan los postes y el larguero de nuestra portería, como así sucede en la figura 4.7(a).

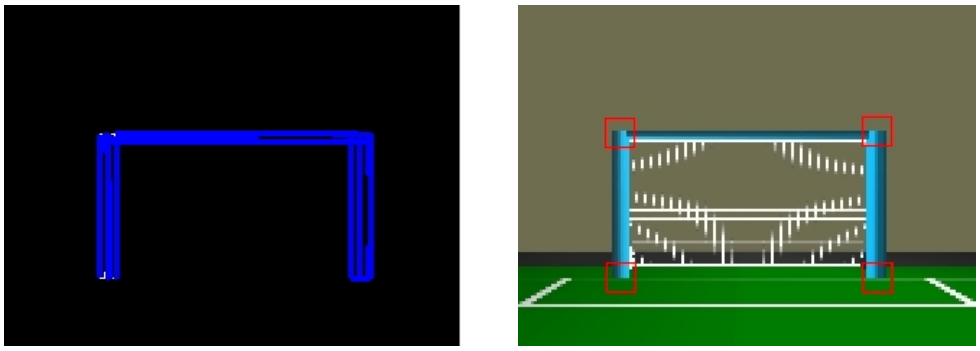
Se utiliza la transformada de Hough probabilística en lugar de la estándar, porque nos da como información dónde comienza y termina cada una de las líneas, al contrario que lo que sucedería con la segunda, ya que la transformada de Hough estándar propaga las líneas detectadas hasta el infinito.

Al igual que con el filtro de color, hemos desarrollado otra herramienta para poder configurar los distintos parámetros que se utilizan en la transformada de Hough, esta herramienta se muestra en la figura 4.5(b).

Cálculo de esquinas

A partir de los extremos de cada línea calculada, necesitamos saber cuáles de esos extremos se corresponden con *Pix1*, *Pix2*, *Pix3* y *Pix4*. Estas esquinas que buscamos se pueden obtener calculando la distancia entre cada extremo y cada esquina de la imagen.

Es decir, para una imagen de dimensiones MxN, calculamos la distancia entre cada extremo calculado y las 4 esquinas de la imagen, que serán los píxeles $(0, 0)$, $(0, N-1)$, $(M-1, 0)$ y $(M-1, N-1)$. Siendo los puntos seleccionados, los 4 extremos que se encuentren a menor distancia de cada una de las 4 esquinas, obteniendo un resultado como el de la imagen 4.7(b).



(a) (b)

Figura 4.7: Transformada de Hough (a) y esquinas finales obtenidas (b).

Como hemos dicho, el filtro Canny y la transformada de Hough sólo se han realizado de momento sobre la imagen del filtro de color azul. Al analizar una imagen como la de la figura 4.7(b), el algoritmo detectaría las 4 esquinas correctamente y no continuaría, puesto que se ha localizado la portería azul. Sin embargo, en el caso de no estar viendo la portería azul, al realizar la transformada de Hough no encontraríamos ninguna línea recta en la imagen.

En este caso, realizaríamos de nuevo el filtro Canny y la transformada de Hough sobre la imagen resultante del filtro de color amarillo. De este modo, si la transformada de Hough encuentra alguna línea, significaría que estamos viendo la portería amarilla, y en caso contrario, no estaríamos viendo ninguna de las dos porterías.

Por lo tanto el algoritmo de detección de la portería es capaz de decirnos si ha encontrado la portería azul, la amarilla o ninguna de ellas, lo que se utilizará posteriormente para situar correctamente al robot con los algoritmos de localización.

4.2.1. Simulador de la portería

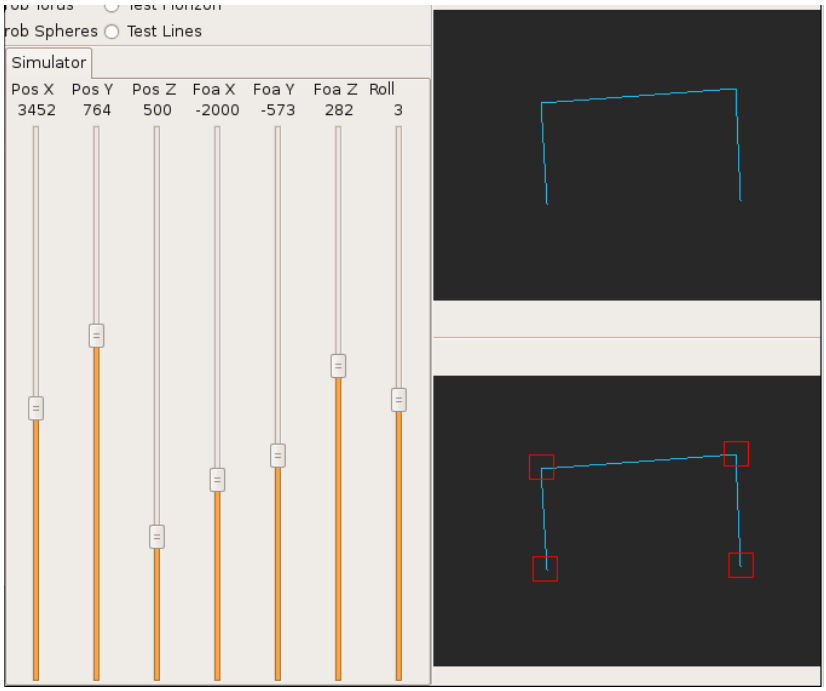
Durante la fase de desarrollo del algoritmo anteriormente descrito vimos la necesidad de probar el funcionamiento de éste de una forma precisa para comprobar que los resultados eran satisfactorios. Por ello, decidimos incorporar dentro de nuestra propia aplicación un simulador con el que trabajar en este aspecto.

En este simulador, representamos una escena en 3D utilizando OpenGL, en la que únicamente aparecía una portería azul con las dimensiones de la portería utilizada en la RoboCup. En esta escena situamos una cámara virtual, simulada también mediante OpenGL, la cual es movible a voluntad mediante una interfaz gráfica de usuario.

Para hacer que nuestro simulador fuese de utilidad, tuvimos en cuenta todos los grados de libertad que podíamos encontrarnos utilizando una cámara Pinhole, que son, como ya describimos en la sección 3.2.1, la posición de la cámara en 3 dimensiones, el foco de atención de la cámara y el roll.

Todos estos parámetros se pueden cambiar de forma sencilla en la interfaz de usuario, permitiendo el movimiento alrededor de la portería en la escena 3D generada y haciendo que podamos probar el algoritmo de una forma rápida.

En la siguiente imagen (figura 4.8), podemos ver una captura de la interfaz generada, mientras detectábamos la portería con el algoritmo que acabamos de describir.



(a)

Figura 4.8: Simulador de la portería empotrado en la aplicación.

4.2.2. Cálculo de horizonte

Con la finalidad de evitar posibles errores en la detección de las porterías, como por ejemplo confundir los colores de uno de los robots con los colores de la portería, hemos generado un algoritmo que se encarga de delimitar dónde termina el campo, de modo que para detectar la portería sólo buscaremos en la imagen fuera de los límites de éste.

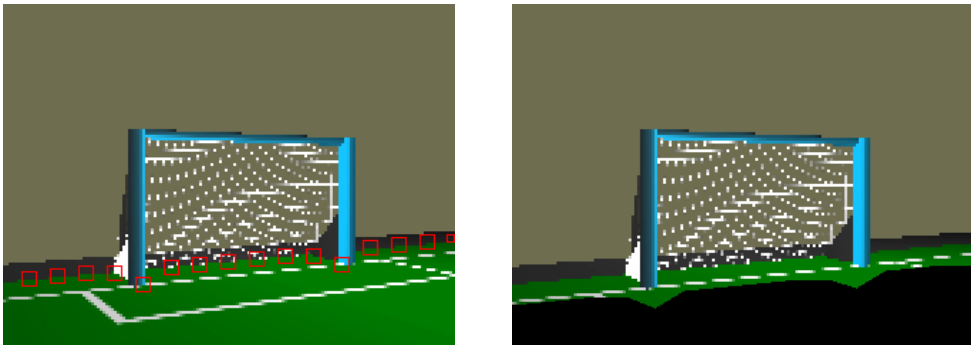
El algoritmo, que recibe como entrada la imagen obtenida por las cámaras, devuelve así una imagen en la que se muestra en negro la parte que queda dentro de los límites del campo calculados.

La forma de saber dónde se termina el terreno de juego, consiste en recorrer cada columna de abajo a arriba comprobando dónde se termina el color verde en esa columna, lo que se realizará utilizamos un filtro de color en HSV similar al utilizado para detectar la portería. Esta función realiza algunas comprobaciones para evitar falsos positivos, como por ejemplo, asegurarse de que el color verde no aparece durante varios píxeles seguidos.

Para que el consumo de recursos del algoritmo no sea muy elevado hemos decido no recorrer todas las columnas que tiene la imagen, sino realizar lo anterior únicamente cada 20 columnas, seleccionando los puntos como pueden verse en el ejemplo de la figura 4.9(a).

En el resto de columnas, se obtiene el límite teniendo en cuenta los puntos seleccionados antes y después de la columna en la que nos encontremos, interpolando linealmente los puntos calculados.

Una vez calculados los píxeles donde termina el campo para cada columna, dejamos un margen de error de 15 píxeles y pintamos de color negro los píxeles que queden por debajo de éstos, devolviendo como resultado una imagen como la que vemos en 4.9(b).



(a) (b)

Figura 4.9: Puntos seleccionados cada 20 píxeles (a). Resultado del algoritmo (b).

4.2.3. Detección de porterías parciales

Como ya hemos dicho, para poder utilizar la detección como entrada en los algoritmos de localización, es necesario que hayamos detectado las 4 esquinas de la portería correctamente. En el caso de ver una portería parcialmente, la detección se realizará de forma similar a como vemos en la figura 4.10.

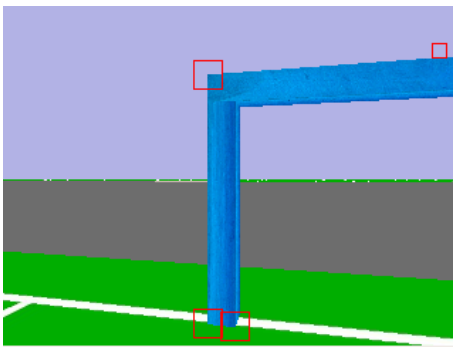
Debido a lo que acabamos de ver, es necesario comprobar cuándo la detección realizada es correcta y cuándo no lo es, para lo que hemos desarrollado una función que se encarga de la comprobación.

Esta función recibe como parámetros de entrada los 4 puntos teóricos de las 4 esquinas de la portería, y a partir de ellos determina si estamos viendo la portería entera o no.

Esto lo conseguimos fijando un margen de error con el que determinamos si dos puntos están lo suficientemente alejados como para considerarlos puntos distintos. En el caso de que alguno de los puntos esté junto a otro (dentro de este margen) la detección no se habrá realizado correctamente, y puede que estemos viendo la portería parcialmente o que no la estemos viendo.

Si todos los puntos están lo suficientemente alejados, se considera que las cuatro esquinas de la portería se han detectado de forma correcta, aunque debido a problemas de oclusiones u otros factores, esto puede no ser del todo cierto.

Antes de comenzar los algoritmos de localización llamamos a esta función, y en el caso de que la detección se establezca como no válida no se continua con la ejecución del algoritmo de localización hasta la siguiente iteración.



(a)

Figura 4.10: Portería vista parcialmente en Gazebo.

4.2.4. Experimentos de detección de portería

Para comprobar el funcionamiento del algoritmo se han realizado numerosas pruebas en busca de casos en los que se pudiesen cometer errores, y que veremos en esta sección.

Las pruebas realizadas tanto de éste como del resto de algoritmos desarrollados han sido realizadas sobre un ordenador Pentium-IV con 3 GHz de frecuencia, lo que deberá ser tenido en cuenta a la hora de evaluar los tiempos de ejecución.

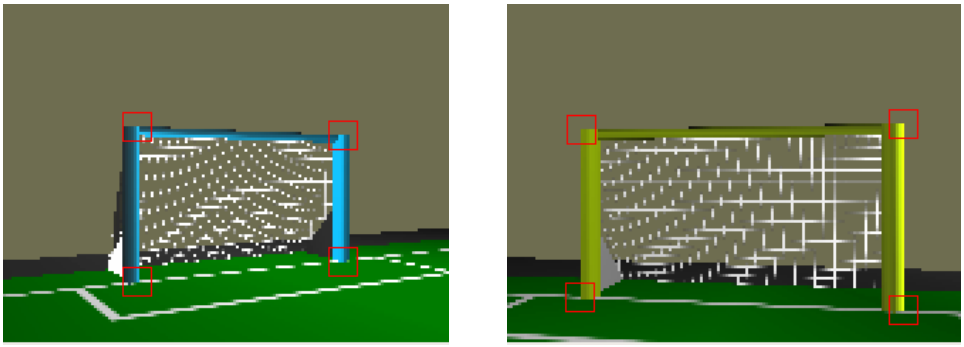
La longitud en líneas de código del algoritmo desarrollado no ha sido muy grande, ya que sólo ha ocupado 170 líneas de código. Además, los tiempos de ejecución son de alrededor de 17 milisegundos, siendo el resultado bastante satisfactorio teniendo en cuenta que las funciones de OpenCV para la detección de bordes y líneas consumen la mayoría del tiempo.

El tiempo de ejecución anterior puede empeorar en el caso de que los filtros de color no funcionen correctamente, puesto que si hay muchos falsos positivos en este filtro de color, tanto la transformada de Hough como el filtro Canny realizarán muchos más cálculos de los necesarios.

A continuación van a mostrarse una serie de imágenes desde distintas posiciones tomadas con los simuladores Webots (4.11) y Gazebo (4.12) para mostrar el funcionamiento del algoritmo. En estas imágenes los colores utilizados en los simuladores para ambas porterías han sido valores por defecto ya almacenados dentro de la aplicación, siendo distintos para cada simulador.

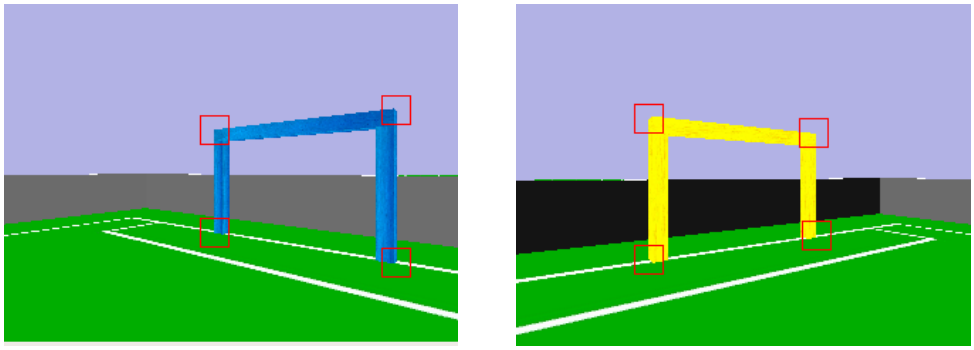
Además de utilizar los simuladores Webots y Gazebo, para avalar la robustez del algoritmo, hemos decidido probar el algoritmo utilizando imágenes capturadas por el robot Nao real, para ello, hemos obtenido una serie de imágenes en la RoboCup disputada en Graz en julio de 2009, apuntando la posición desde la que se tomaban. Aquí vemos algunos ejemplos utilizando estas imágenes reales (4.13).

El balance general del algoritmo es muy positivo, puesto que es muy robusto e incluso es independiente a los cambios en la orientación de la cámara, que pueden producirse en la realidad al girar la cabeza el robot. La precisión en la detección depende de la distancia que haya a la portería, siendo peor esta precisión conforme más nos alejamos, y de los parámetros del filtro de color y la transformada de Hough, haciendo que la detección sea incorrecta cuando los parámetros no estén bien configurados.



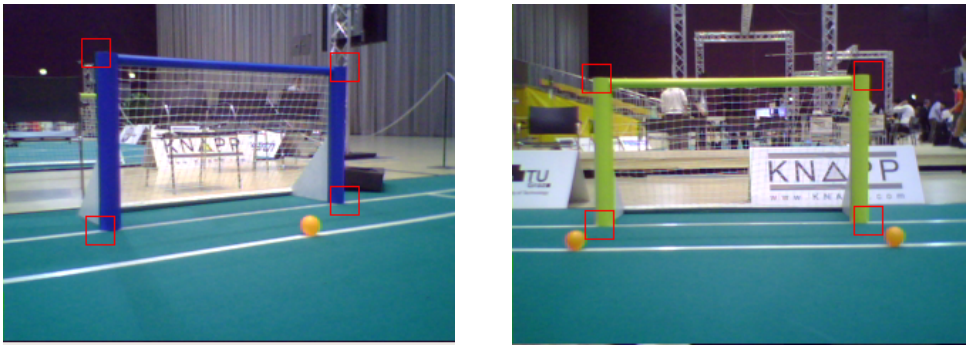
(a) (b)

Figura 4.11: Detección de portería en Webots



(a) (b)

Figura 4.12: Detección de portería en Gazebo



(a) (b)

Figura 4.13: Detección de portería con imágenes reales.

Alternativas en la detección de la portería

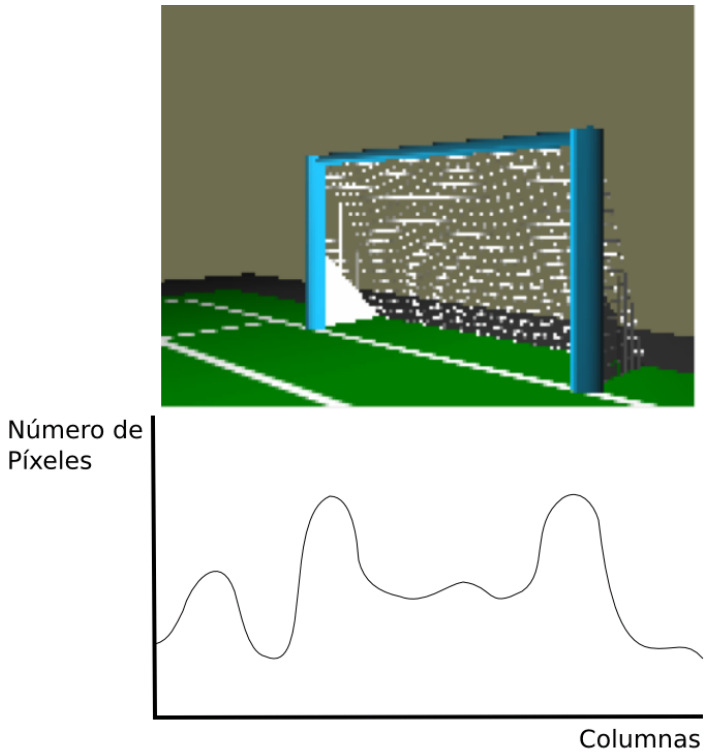
Además del algoritmo que se ha descrito anteriormente, realizamos otro algoritmo de detección para la portería utilizando la segmentación por umbrales.

La segmentación es un proceso que consiste en dividir una imagen en regiones homogéneas que tengan una o varias características en común. La técnica que hemos empleado para obtener las regiones segmentadas es la utilización de histogramas de color.

Un histograma es un diagrama de barras que contiene frecuencias relativas a alguna característica, en nuestro caso, el color. Para la realización de nuestro algoritmo primero realizamos un histograma por columnas identificando para cada columna cuántos píxeles pasan un filtro de color HSV relacionado con el color de la portería.

Este filtro de color, al igual que el del algoritmo ya comentado, conocía el color a buscar en la imagen seleccionando los valores HSV a través de una interfaz gráfica de usuario.

El objetivo de este primer histograma por columnas es limitar el ancho de la portería, con lo que obtendremos un histograma parecido al de la figura 4.14

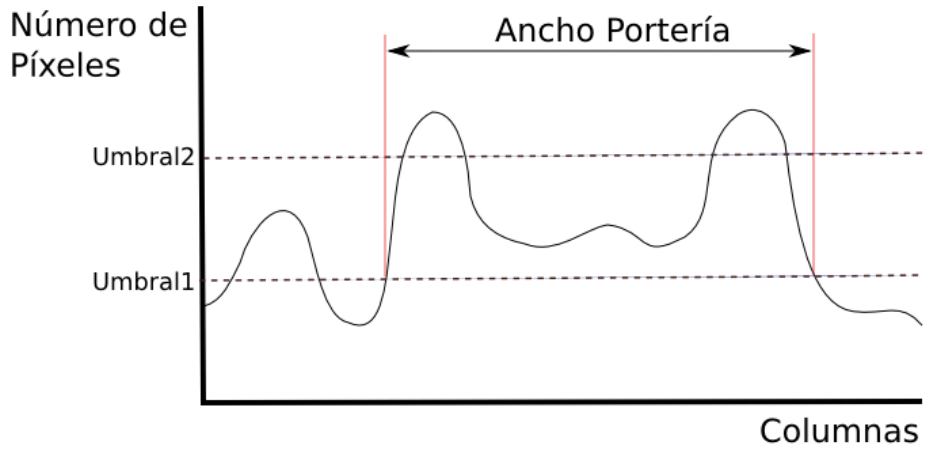


(a)

Figura 4.14: Histograma por columnas.

Para delimitar dónde comienza y termina la portería a partir del histograma obtenido, una posible técnica sería la utilización de un umbral simple, donde se consideraría que la portería se encuentra en las columnas donde el valor estuviese por encima del umbral. Sin embargo este tipo de umbrales pueden dar muchos problemas, ya que un umbral demasiado bajo podría dar falsos positivos, mientras que uno demasiado alto, podría dar falsos negativos. Por ello, y basándonos en el proyecto [Martínez de la Casa, 2003], decidimos utilizar un doble umbral.

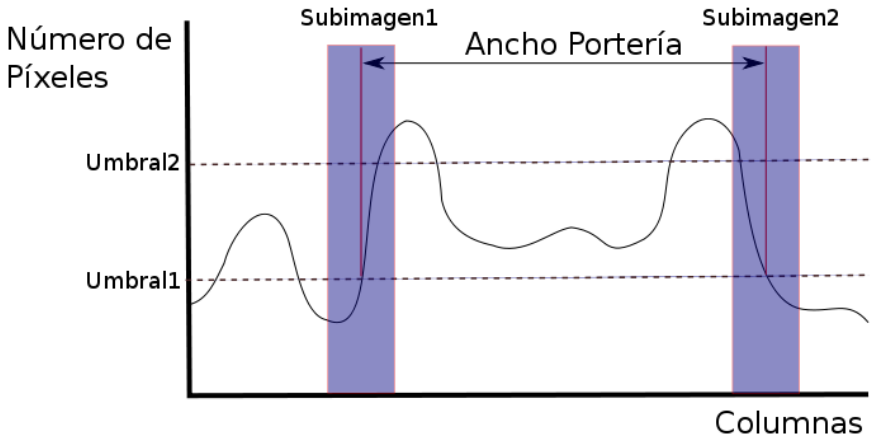
Por tanto, para obtener el ancho de la portería, apuntamos la columna donde se sobrepasa el primer umbral, siendo el final de la portería el lugar donde se pasa por debajo de este primer umbral. Para considerar a esta zona válida, se debe superar en algún momento el segundo umbral, si no se considerará que se trata de un falso positivo. Finalmente, si obtenemos varias zonas posibles que hayan superado ambos umbrales, y teniendo en cuenta que la portería sólo puede encontrarse en uno de ellos, nos quedamos con la zona con mayor número de columnas. Al aplicar este doble umbral a la figura 4.14, obtendríamos algo similar a la figura 4.15.



(a)

Figura 4.15: Histograma por columnas tras aplicar dos umbrales.

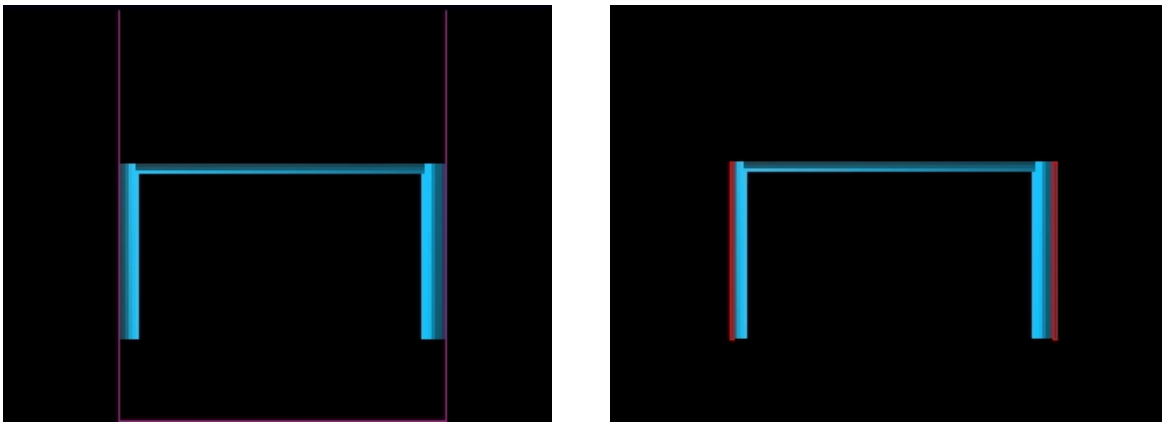
Una vez localizada la portería en la imagen, podemos deducir que los extremos de la zona seleccionada corresponden con el poste derecho e izquierdo de la portería. Para conocer cuál es la altura de cada uno de los postes, para cada postes obtenemos una subimagen desde $columna - 10$ hasta $columna + 10$, como se refleja en la figura 4.16.



(a)

Figura 4.16: Subimagen seleccionada para cada poste.

Para estas dos subimágenes seleccionadas, realizamos un histograma por filas siguiendo el mismo procedimiento que el utilizado anteriormente con el histograma de columnas. Con esto obtendremos la altura de cada uno de los postes observando dónde termina y acaba cada poste, con lo que finalmente obtendríamos las 4 esquinas que buscábamos. Este proceso se puede ver sobre una imagen real en la figura 4.17, donde primero obtenemos el ancho de la portería y después la altura de cada poste.



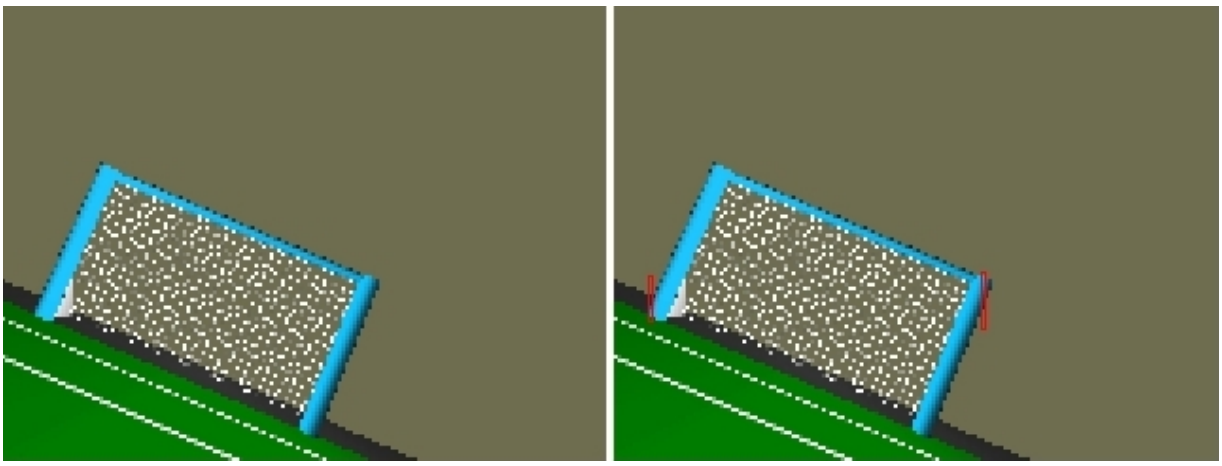
(a) (b)

Figura 4.17: Ancho de la portería (a). Resultado final (b)

Al realizar diversas pruebas sobre este algoritmo, comprobamos que en la mayoría de las ocasiones la detección de la portería se realizaba correctamente, y además las complejidades en tiempo y espacio eran similares a las obtenidas con el algoritmo visto en 4.2.

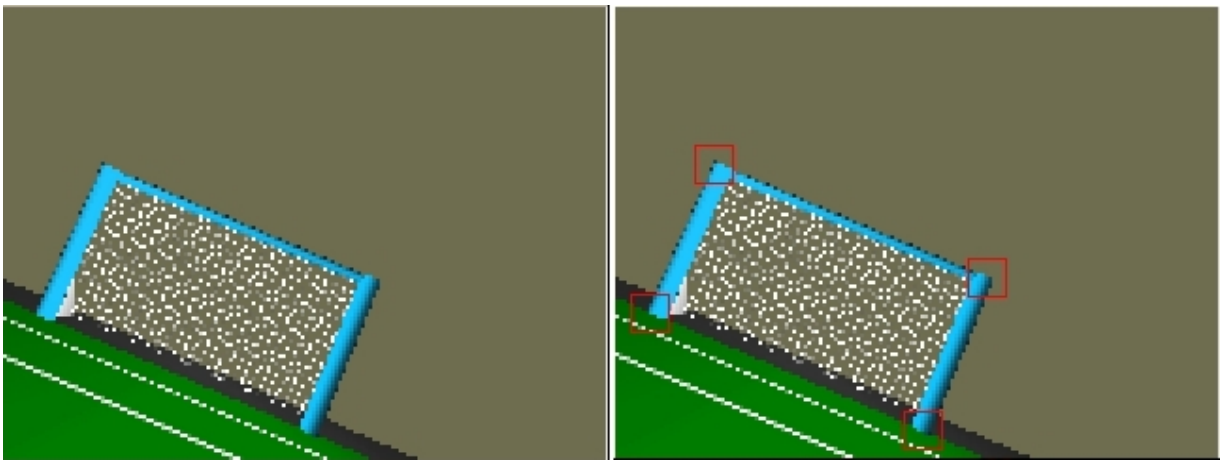
Sin embargo, nos dimos cuenta de que había algunos problemas en la detección cuando la cámara utilizada estaba ligeramente girada. Cuando teníamos imágenes similares a la mostrada en la figura 4.18 el resultado obtenido no era satisfactorio, puesto que las alturas de los postes no se calculaban bien debido a la inclinación. Por el contrario, al utilizar el algoritmo visto anteriormente, el resultado era el correcto (figura 4.19).

Decidimos desechar este algoritmo puesto que no nos aseguraba robustez en algunas ocasiones, y utilizamos como algoritmo final el primeramente detallado en este capítulo.



(a)

Figura 4.18: Imagen original y postes detectados utilizando doble umbral.



(a)

Figura 4.19: Imagen original y postes detectados utilizando Canny + Hough.

4.3. Localización instantánea 3D usando toros

El primer algoritmo desarrollado para la localización instantánea toma como entrada los 4 puntos (píxeles) obtenidos con el algoritmo de detección de porterías. Para cada uno de estos puntos, utilizamos la función *backproject* de la librería Progeo de JdeRobot, como se comentó en 3.2.1, obteniendo una recta proyectiva en 3D de cada punto en 2D.

Esta primera técnica se basa en el ángulo con el que vemos los postes de la portería en la imagen. En el momento en el que vemos un poste en la imagen cuyos extremos forman un ángulo determinado, sólo podemos encontrarnos en ciertos lugares del campo para ver a ese poste con ese ángulo. Conociendo además las propiedades del modelo de cámara que estamos utilizando, se producen algunas restricciones geométricas de la posición de la cámara según lo que estemos observando.

Sabiendo lo anterior, si tenemos en cuenta solamente el plano que forma la cámara con los dos extremos del poste, el lugar geométrico en 2D en el que nos encontramos vendrá dado por el arco capaz, y formará un arco de circunferencia que pasará por los dos extremos del poste.

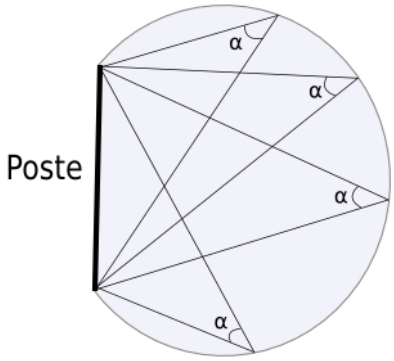
Si revolucionamos la circunferencia completa respecto al eje del poste, obtendremos una figura geométrica en 3 dimensiones que se conoce con el nombre de toro de revolución, que indicará todas las posibles posiciones en las que se encuentra la cámara para ver el poste con ese ángulo.

A partir de aquí, utilizaremos las rectas proyectivas de *Pix1-Pix3* (poste derecho) y *Pix2-Pix4* (poste izquierdo) por separado para hacer los cálculos geométricos necesarios que nos llevarán a calcular primero el arco capaz de cada poste y después el toro en 3D.

Cálculo del toro en un solo poste

Con la información del poste de la que disponemos, es decir, las rectas proyectivas 3D de los extremos superior e inferior, el primer objetivo del algoritmo es calcular el ángulo en el espacio 3D que forman estas dos rectas.

Conociendo la longitud del segmento (en este caso el poste), el arco capaz se define como el lugar geométrico de los puntos desde los cuales se ve al segmento con esa longitud en la imagen, o lo que es lo mismo, los puntos donde los extremos del segmento se ven con un ángulo α determinado. Si dibujamos todos estos puntos obtendremos un arco de circunferencia que pasará por los dos extremos del segmento, como se puede ver en la figura 4.20.



(a)

Figura 4.20: Arco de circunferencia formada por los puntos que ven el poste con ángulo α.

Teniendo en cuenta lo anterior, el ángulo que forman nuestras rectas se corresponderá con el ángulo α utilizado para calcular el arco capaz. Es decir, si calculamos la circunferencia que se forma para ese ángulo α usando el arco capaz, sabemos que nuestra cámara se encontrará en el perímetro de esta circunferencia, lo que sería un primer paso para localizarnos.

Como conocemos las rectas proyectivas 3D de los extremos del segmento, podemos calcular el ángulo α utilizando la ecuación del producto escalar de dos vectores (ecuación 4.1), y despejando para obtener la ecuación 4.2, donde u y v son los vectores correspondientes a las rectas proyectivas y α es el ángulo entre los vectores.

$$uv = |u||v|cos(\alpha) \tag{4.1}$$

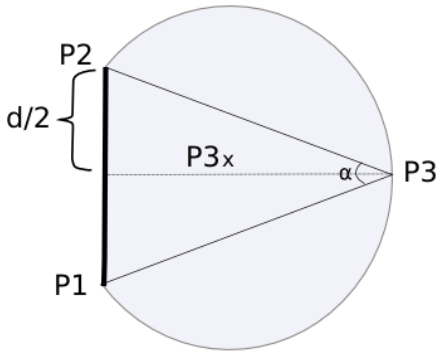
$$\alpha = arccos(\frac{uv}{|u||v|}) \tag{4.2}$$

El siguiente paso a realizar es calcular la circunferencia en 2D vista anteriormente sobre un plano perpendicular al suelo y por el que pasen tanto el poste como las rectas proyectivas.

Una circunferencia en 2D tiene como ecuación la que vemos en 4.3, donde (X, Y) son las coordenadas de un punto cualquiera de la circunferencia, (P_x, P_y) es el centro de ésta y r es su radio. Para despejar las incógnitas X, Y y r, necesitamos 3 puntos, pero en principio sólo conocemos 2, que son los extremos del poste en coordenadas absolutas.

$$(X - P_x)^2 + (Y - P_y)^2 = r^2 \tag{4.3}$$

El tercer punto que necesitamos para resolver las ecuaciones, podría ser cualquiera de los puntos que forman la circunferencia. Para simplificar los cálculos, vamos a calcular el punto en 2D que se encuentra en el plano vertical de la propia portería, y cuya altura es la mitad de la del poste (figura 4.21).



(a)

Figura 4.21: Cálculo del tercer punto de la circunferencia.

Para obtener este tercer punto que acabamos de ver, se utilizan las ecuaciones 4.4 y 4.5, donde se conocen las coordenadas absolutas de los extremos de los postes (*P1* y *P2*), así como la longitud del poste d y el ángulo α, con lo que obtendremos las coordenadas x (horizontal) e y (vertical) del tercer punto.

$$tan(\frac{\alpha}{2}) = \frac{\frac{d}{2}}{P3_x} \rightarrow P3_x = \frac{\frac{d}{2}}{tan(\frac{\alpha}{2})} \tag{4.4}$$

$$P3_y = \frac{P1_y + P2_y}{2} \tag{4.5}$$

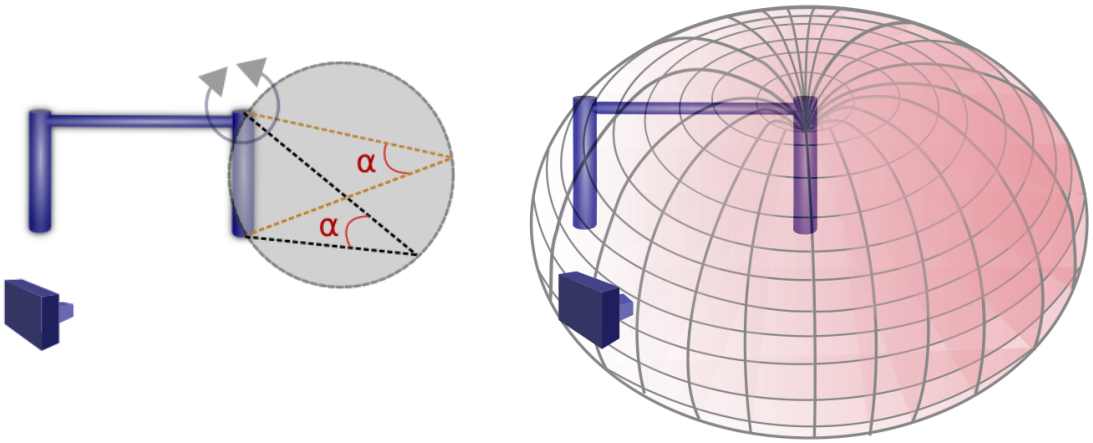
Una vez calculados 3 puntos de la circunferencia, podemos despejar las incógnitas mediante las ecuaciones 4.6.

$$\begin{aligned} (P1_x - X)^2 + (P1_y - Y)^2 &= r^2 \\ (P2_x - X)^2 + (P2_y - Y)^2 &= r^2 \\ (P3_x - X)^2 + (P3_y - Y)^2 &= r^2 \end{aligned} \tag{4.6}$$

Cabe destacar, que los cálculos que acabamos de realizar son independientes de la inclinación de la cámara, por tanto son insensibles al giro que se pueda producir cuando ande el robot. Los puntos válidos en los que puede encontrarse la cámara, son un

subconjunto de esa circunferencia, ya que en la parte de la circunferencia que no hemos dibujado en 4.20, el ángulo en grados sería 180 - α.

Cualquier otro plano vertical que contenga al poste contendrá otro arco capaz donde podría estar la cámara real. El conjunto de arcos capaces válidos en 3D se puede calcular revolucionando la circunferencia que hemos calculado alrededor del eje del propio poste, como se ve en la figura 4.22(a) y con ello obtendremos la figura geométrica conocida como toro (4.22(b)). Este toro contendrá todas las posibles posiciones en las que nuestro robot puede encontrarse para ver al poste con α grados, sin importar la orientación de la cámara.



(a) (b)

Figura 4.22: Rotación de la circunferencia sobre el eje Z (a). Toro generado en el poste derecho (b).

La ecuación general que define al toro es 4.7, donde X es la profundidad, Y la horizontal, Z la vertical, R es la distancia entre el centro del toro y el centro de la circunferencia y r es el radio de la circunferencia.

Los parámetros R y r, ya son conocidos de cuando se calculó la circunferencia en 4.6, siendo r el mismo radio que el de la circunferencia calculada, y correspondiendo el valor de R con la coordenada horizontal del centro de la circunferencia en esa ecuación (X).

Ahora el objetivo es poder despejar los parámetros (X, Y, Z) que cumplan la ecuación del toro y que nos indicarían la posición absoluta de la cámara. Sin embargo, sólo disponemos de una ecuación y de tres incógnitas, por lo que por el momento no podemos despejarlos.

$$(R - \sqrt{X^2 + Y^2})^2 + Z^2 = r^2 \tag{4.7}$$

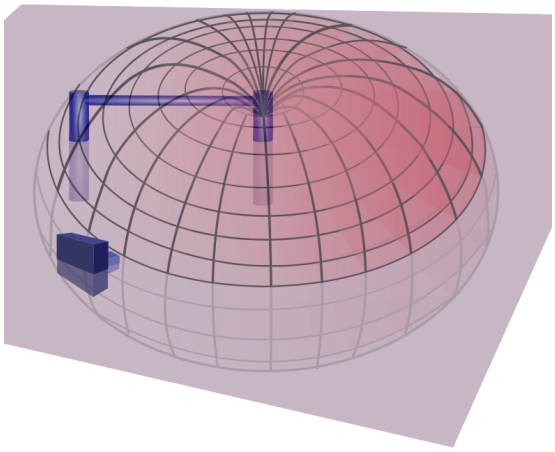
Cálculo de la posición usando los dos postes

Por el momento, hemos podido calcular la ecuación general de un toro, pero nos quedan 3 incógnitas por despejar para obtener la posición final. En principio, necesitaríamos otras dos ecuaciones similares para poder despejar la posición en 3D, que podrían obtenerse utilizando el otro poste de la portería y el larguero.

Sin embargo, pudimos comprobar que la resolución analítica de estas ecuaciones tenían una gran complejidad, puesto que a pesar de existir solución, ésta era múltiple y era computacionalmente muy lenta.

Por ello, decidimos simplificar las ecuaciones teniendo en cuenta la altura del robot, puesto que es conocida (50 centímetros). Así, la tercera ecuación que necesitamos la hemos obtenido utilizando un plano paralelo al suelo con la misma altura que la del robot. Este plano lo intersecamos con el toro calculado (figura 4.23), lo que nos da como resultado dos circunferencias, una de la zona interna del toro y otra de la externa.

Como ya vimos en la figura 4.20, sólo nos interesa uno de los arcos de la circunferencia, el interno o el externo, dependiendo del caso. Podemos saber cuál de las dos circunferencias nos interesa sabiendo el ángulo que formaban las rectas proyectivas del poste, seleccionando la circunferencia interior cuando el ángulo α es mayor de 90 grados, y la exterior en otro caso.



(a)

Figura 4.23: Intersección de un toro con un plano.

Para calcular las circunferencias en 2 dimensiones que se obtienen al intersecar el plano con el toro, despejamos la ecuación general del toro 4.7 como en la ecuación 4.8, donde las 3 variables de la parte derecha de la ecuación son conocidas, R y r por haber sido calculadas anteriormente, y Z por ser la altura del plano que intersecamos (la altura del robot).

$$X^2 + Y^2 = (R - \sqrt{r^2 - Z^2})^2 \tag{4.8}$$

Además, teniendo en cuenta la ecuación general de la circunferencia en 2D vista en 4.3, podemos observar como la parte izquierda de las ecuaciones 4.8 y 4.3 coincidirían si el punto central (P_x, P_y) fuese $(0, 0)$, lo que es cierto desde un sistema de coordenadas solidario con el poste, obteniendo la ecuación 4.9.

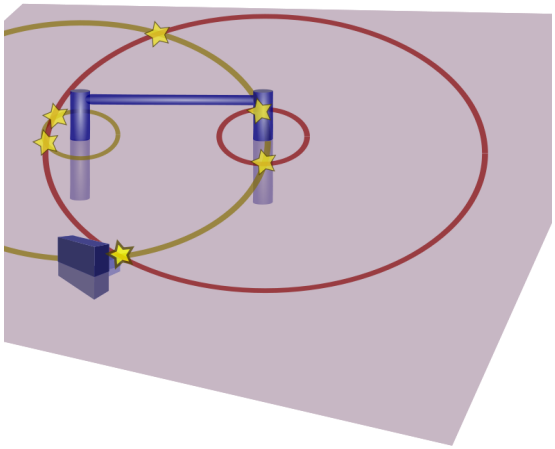
Por lo tanto nuestras circunferencias, tendrán como radio el resultado de la parte derecha de la ecuación 4.8 (quedándonos con el radio de la raíz positiva o negativa dependiendo de α) y su centro estará dado por la posición conocida del poste en coordenadas absolutas.

$$X^2 + Y^2 = R^2 \tag{4.9}$$

Al calcular la circunferencia de cada poste, podemos obtener nuestra posición en coordenadas absolutas intersecando las dos circunferencias como en la figura 4.24).

Este cálculo final se realiza con las ecuaciones de las dos circunferencias (4.10), teniendo en cuenta que ambos postes tienen coordenadas en 3D $(Pi_x, 0, Pi_z)$, puesto que así se ha establecido en el sistema de coordenadas absolutas, y no siendo necesaria la coordenada z para el cálculo de la posición, al ser siempre la altura del robot.

$$\begin{aligned} (X - P1_x)^2 + Y^2 &= R1'^2 \\ (X - P2_x)^2 + Y^2 &= R2'^2 \end{aligned} \tag{4.10}$$



(a)

Figura 4.24: Cálculo de la posición final.

4.3.1. Experimentos

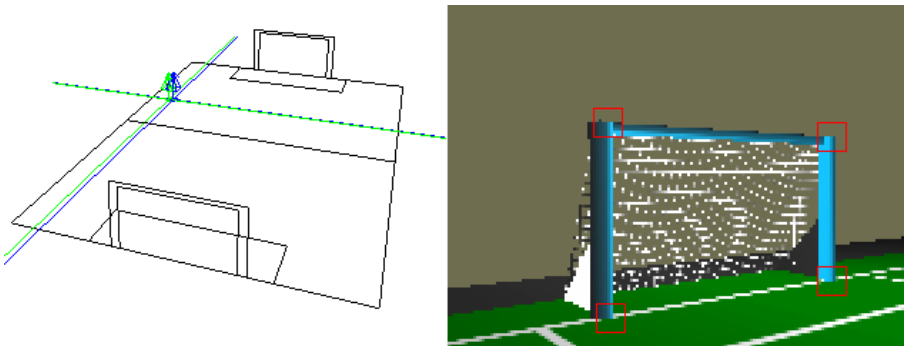
El algoritmo que acabamos de describir ha ocupado un total de 100 líneas de código y tiene un tiempo medio de ejecución de unos 19 milisegundos, donde 17 de ellos se corresponden con el tiempo de ejecución de la detección de la portería, por lo que tan solo 2 milisegundos corresponden al propio algoritmo.

A continuación vamos a mostrar varias imágenes de la portería azul tomadas en el simulador Webots en cuatro posiciones distintas, que nos permitirán estimar el error medio del algoritmo de modo aproximado (figura 4.25). Durante las pruebas realizadas con el algoritmo, se han realizado un gran número de experimentos, pero hemos decidido mostrar estas imágenes puesto que eran bastante representativas del resto de casos.

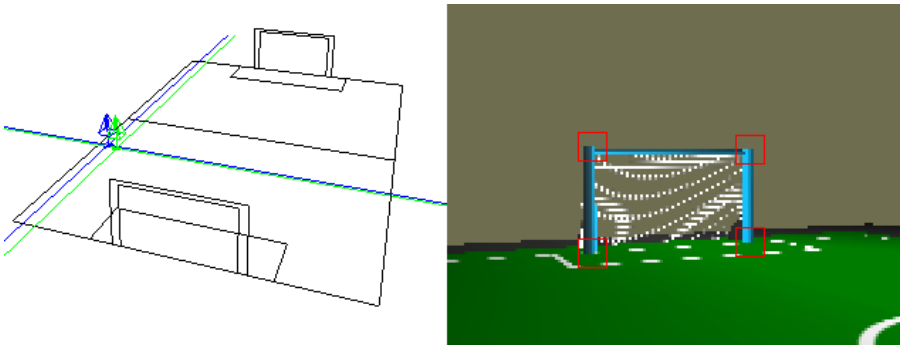
En las 4 imágenes mostradas, podemos ver dos flechas que indican posiciones, la flecha verde indica siempre la posición real en la que nos encontramos, mientras que la flecha azul es la calculada por el algoritmo, de esta forma podemos ver el error cometido en el cálculo de forma visual.

Para el caso que acabamos de ver, el error medio de las 4 imágenes tomadas ha sido de 10.5 centímetros, lo que nos permite ver que tiene una gran precisión.

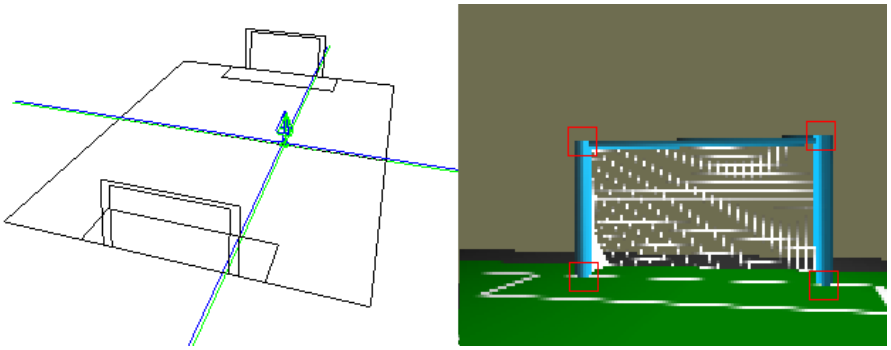
Si en la imagen lo que vemos es la portería amarilla, el algoritmo actuará de la misma forma que si se tratase de la portería azul, sin embargo, como el algoritmo que detecta la portería nos indica cuál de las dos porterías estamos viendo, a la hora de guardar la posición



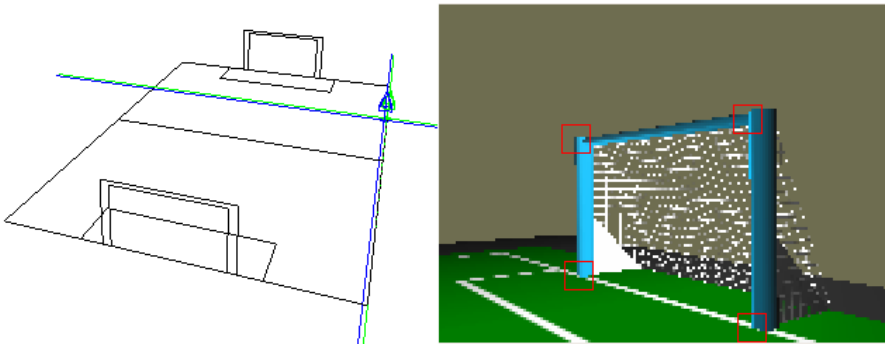
(a)



(b)



(c)

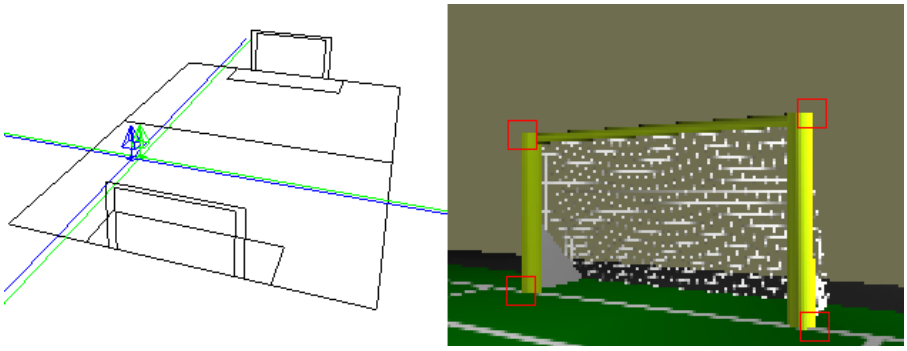


(d)

Figura 4.25: Localización instantánea usando toros en Webots.

final calculada modificaremos las coordenadas x (profundidad) e y (ancho) teniendo en cuenta las dimensiones del campo, para situarnos en la zona contraria del campo.

En la imagen 4.26 podemos ver cómo se calcula la posición sobre la portería amarilla y se muestra la localización correcta.



(a)

Figura 4.26: Localización instantánea usando toros con portería amarilla.

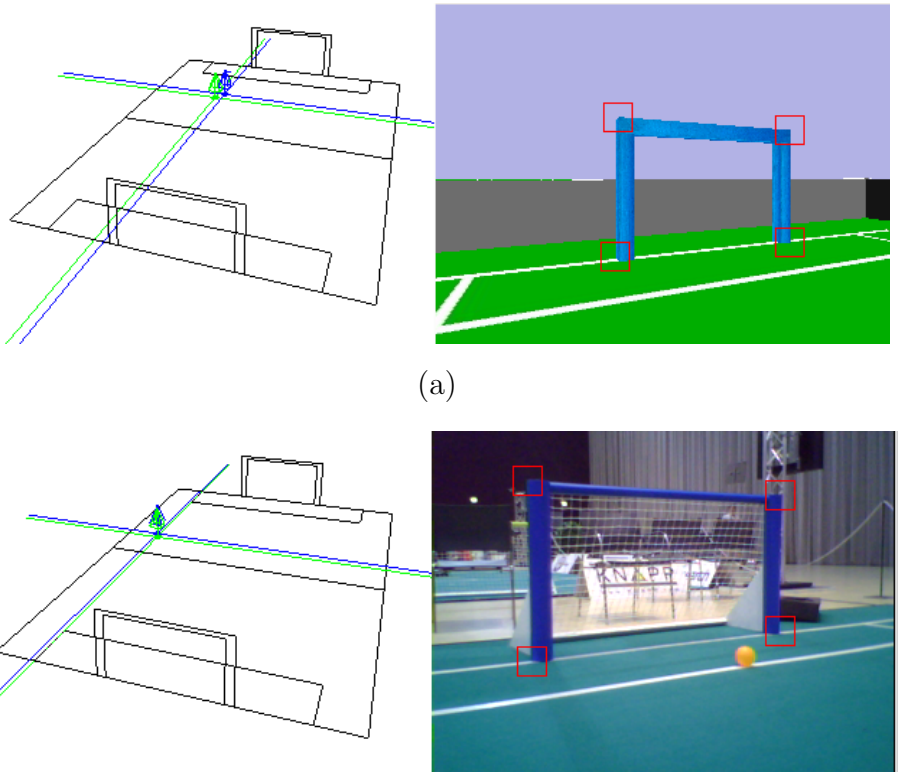
Las imágenes anteriores han mostrado el comportamiento del algoritmo en el simulador Webots, sin embargo, podemos comprobar como el funcionamiento sigue siendo muy preciso en el simulador Gazebo 4.27(a) e incluso utilizando imágenes reales 4.27(b).

Este algoritmo puede tener errores cuando la detección de la portería no esté correctamente realizada, un caso de lo anterior puede verse en la figura 4.28, donde al establecer unos parámetros para el filtro de color incorrectos, la portería no se detecta correctamente y por ello la posición calculada no es la adecuada.

En general, el algoritmo tiene un comportamiento bastante bueno en todo el campo, funcionando de manera similar tanto en los laterales como en la parte central de éste. No obstante, al aumentar la distancia respecto a la portería, la precisión obtenida va siendo progresivamente menor, también influido por el hecho de que la detección de la portería es menos precisa.

La sensibilidad con respecto a la detección de la portería es baja, no influyendo de una manera notable un error de pocos píxeles en la detección de ésta, aunque la influencia puede agravarse al aumentar la distancia, como ya hemos dicho.

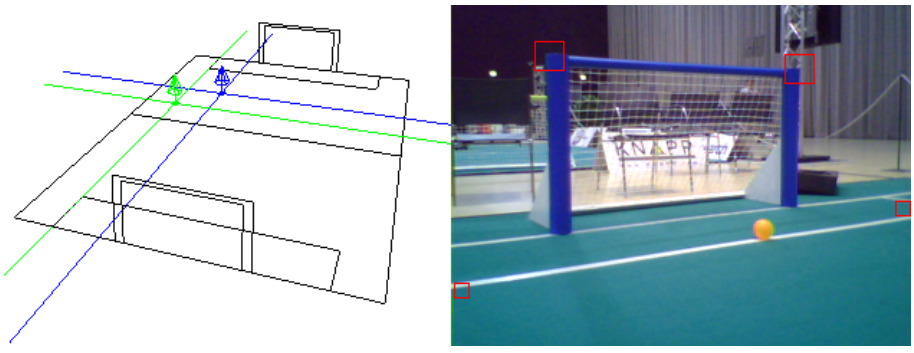
El punto negativo del algoritmo es la necesidad de conocer la altura del robot, algo que en la mayoría de las ocasiones no es posible porque puede variar a lo largo de la ejecución, aunque en el caso del robot Nao siempre podremos suponer un rango de alturas en el que se encontraría el robot.



(a)

(b)

Figura 4.27: Imagen utilizando el simulador Gazebo (a) e imagen real tomada del Nao (b).



(a)

Figura 4.28: Error en la localización instantánea usando toros.

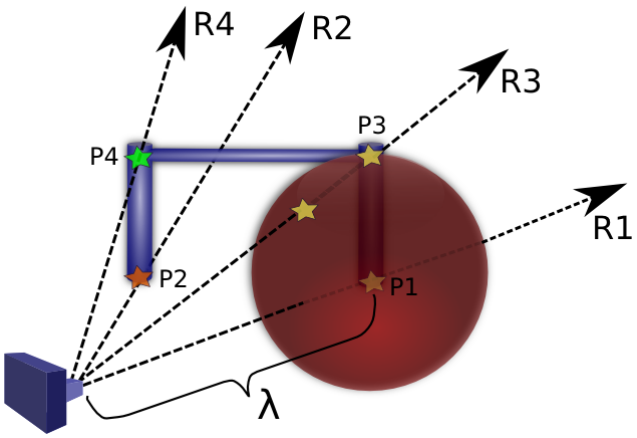
4.4. Localización instantánea 3D usando esferas

El segundo método de localización instantánea, al igual que el anterior, utiliza como entrada las 4 esquinas de la portería y calcula con Progeo las rectas proyectivas de cada esquina de la portería.

En este caso, el algoritmo basa su planteamiento en la posición de los vértices de la portería en la imagen. El hecho de ver que las esquinas de la portería se encuentran en una posición determinada, hace que se produzcan de nuevo restricciones geométricas sobre la posición de la cámara, puesto que sólo pueden verse las esquinas de la portería en esas posiciones desde determinados lugares del campo.

El algoritmo consiste en elegir una de las 4 esquinas de la portería y a partir de esta esquina, conocer la posición real del resto de esquinas teniendo en cuenta las distancias reales entre ellas y los píxeles de la imagen en los que se han detectado.

Para ello, desde la esquina elegida se generan 3 esferas (una por cada esquina restante), en las que cada esfera tiene como radio la distancia en la realidad entre su centro y la esquina que le corresponde. Cada una de estas esferas se interseca con el rayo proyectivo de la esquina que le corresponde, obteniendo al final varias posibles posiciones entre las que se encontrará la posición real. Este método se muestra visualmente en la figura 4.29.



(a)

Figura 4.29: Intersección de una esfera con su rayo proyectivo a una distancia concreta (λ).

Si supiésemos la posición en la que se encuentra una de las esquinas respecto a la cámara, lo que acabamos de comentar sería muy fácil de realizar, y podríamos obtener las posiciones de todas las esquinas de la portería respecto a la cámara de forma trivial.

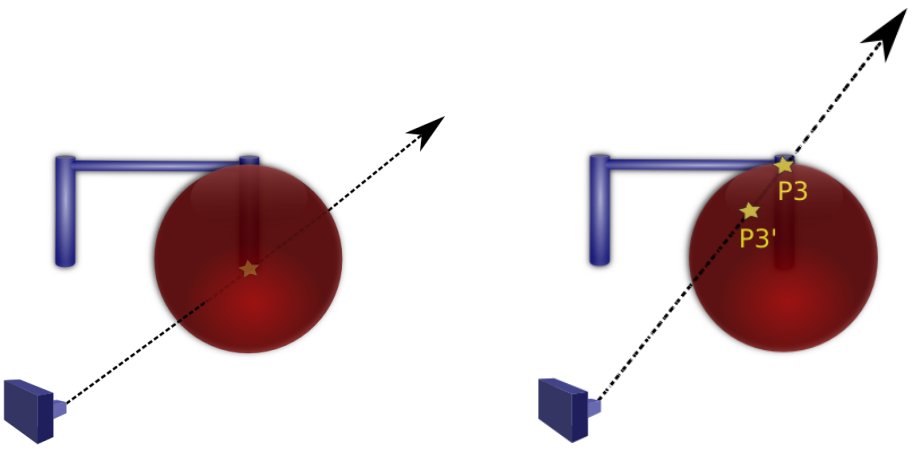
Sin embargo, como no conocemos dónde se encuentra ninguna esquina, sino que sólo conocemos sus rayos proyectivos, vamos a suponer cada distancia posible (a la que llamaremos λ) dentro de una de las rectas proyectivas, y le asociaremos una función de coste, lo que nos hará plantear la localización como un problema de optimización.

Cálculo de la posición con un punto conocido

El algoritmo toma como base la recta proyectiva de *Pix1*, a la que llamaremos *R1*, donde para simplificar la explicación, vamos a suponer que conocemos la posición de la esquina de la portería *P1* dentro de *R1*.

El primer paso consiste en calcular dónde se encontraría *P3* para esa posición de *P1*. Para ello, como conocemos la distancia desde *P1* a *P3* (la altura del poste derecho), generamos una esfera que tenga como radio esta distancia *P1-P3*, para obtener lo que vemos en la figura 4.30(a).

Esta esfera se interseca con la recta proyectiva de *Pix3*, lo que nos dará como resultado normalmente dos puntos candidatos, *P3* y *P3'* (como se muestra en la figura 4.30(b)), aunque puede darnos tan solo uno en caso de que la recta sea tangente a la esfera o ninguno cuando no la atraviese.



(a) (b)

Figura 4.30: Rayo proyectivo de *Pix1* (a). Puntos obtenidos en la intersección (b).

Para encontrar los puntos donde interseca la recta con la esfera generada, hemos hecho uso de la ecuación continua de la recta (4.11) y de la ecuación general de la esfera (4.12). Donde (x_0, y_0, z_0) es el punto origen de la cámara en coordenadas relativas, es decir (0,

0, 0), el vector director (V_x, V_y, V_z) es la recta proyectiva de *Pix3*, (P_x, P_y, P_z) es el punto seleccionado sobre *R1* y R es la distancia conocida entre los dos puntos.

$$\frac{X - x_0}{V_x - x_0} = \frac{Y - y_0}{V_y - y_0} = \frac{Z - z_0}{V_z - z_0} \tag{4.11}$$

$$(X - P_x)^2 + (Y - P_y)^2 + (Z - P_z)^2 = R^2 \tag{4.12}$$

Por lo tanto sustituyendo (X, Y, Z) de 4.11 en 4.12 obtenemos la ecuación 4.13, con la que podemos calcular la coordenada Z de la intersección, y a partir de ella las coordenadas X e Y de forma trivial con la ecuación 4.11.

En el caso de que Z no tenga soluciones reales, la recta y la esfera no intersecarán en ningún punto, y no se habrá encontrado *P3*.

$$(\frac{V_x * Z}{V_z} - P_x)^2 + (\frac{V_y * Z}{V_z} - P_y)^2 + (Z - P_z)^2 = R^2 \tag{4.13}$$

Una vez que hemos calculado los puntos candidatos para ser *P3*, realizaríamos el mismo procedimiento con *P2* y *P4*, para lo cual sólo tendríamos que generar esferas con los radios correspondientes, que serían el ancho de la portería en el caso de *P1-P2*, y la diagonal para *P1-P4*. Así tendríamos que calcular las intersecciones de cada recta proyectiva con su esfera correspondiente.

Realizando lo anterior, para una sola posición conocida de *P1*, habríamos encontrado en el peor de los casos 8 posibles combinaciones de puntos, ya que para *P2*, *P3* y *P4* podemos encontrar 2 puntos candidatos, teniendo entonces *P2'*, *P3'* y *P4'*.

A estas 8 combinaciones, las sometemos a una función de coste, que estará basada en propiedades geométricas de la portería como veremos más adelante, y nos quedaremos como "ganador" con la combinación que menos coste tenga.

Cálculo de la posición sin conocimiento previo

En la sección anterior, partíamos de la base de que conocíamos dónde se encontraba *P1* dentro de la recta proyectiva de *Pix1*. Sin embargo, este conocimiento no es real, así que debemos tener en cuenta los infinitos puntos que componen *R1* y que son candidatos a ser *P1*.

La solución adoptada, consiste en recorrer la recta proyectiva *R1*, eligiendo distintas distancias a las que hemos llamado λ, como ya se mostró en la imagen 4.29.

Para cada posible posición de *P1* realizamos lo descrito en la sección anterior, quedándonos siempre con un "ganador". Para tener suficientes λ, hemos recorrido la recta proyectiva avanzando un centímetro cada vez y finalizando el algoritmo al llegar a una distancia prefijada (8 metros) o cuando alguna de las rectas proyectivas no interseca con su esfera (las raíces son negativas).

Además, en cada iteración se comprueba si se ha mejorado el coste respecto a otras iteraciones, para guardar siempre el mejor ganador de todas las iteraciones y el mejor coste obtenido, para al final quedarnos tan solo con una posición entre todos los candidatos.

El algoritmo en pseudocódigo por tanto sería:

```
rpR1 = Recta Proyectiva Pix1
rpR2 = Recta Proyectiva Pix2
rpR3 = Recta Proyectiva Pix3
rpR4 = Recta Proyectiva Pix4
radio1 = Distancia(Pix1, Pix2); //Ancho de la portería
radio2 = Distancia(Pix1, Pix3); //Altura del poste
radio3 = Distancia(Pix1, Pix4); //Diagonal
mejorcoste = infinito;
Foreach (1cm in rpR1 -> punto1) {
    (p2, p2') = calcularInterseccion(punto1, radio1, rpR2);
    (p3, p3') = calcularInterseccion(punto1, radio2, rpR3);
    (p4, p4') = calcularInterseccion(punto1, radio3, rpR4);
    if not hayIntersecciones(p2, p2', p3, p3', p4, p4')
        finalizar;
    (coste, ganador) = calcularCoste(p1, p2, p2', p3, p3', p4, p4');
    if(coste < mejorcoste) {
        mejorcoste = coste
        mejorpunto = ganador
    }
}
```

A pesar de que el algoritmo anterior funciona correctamente, decidimos mejorar la robustez de éste realizando los mismos cálculos que hemos visto sobre *R1*, pero utilizando la recta proyectiva de *Pix2* (poste izquierdo), llamada *R2*, con lo que hemos obtenido mayor precisión a la hora de situar al robot en el campo.

Además, para no pasar a tener 16 candidatos para cada λ, 8 por cada recta proyectiva, decidimos quedarnos tan solo con 4 candidatos para cada una de las rectas, puesto que muchas de ellas estaban repetidas. Así, las combinaciones repetidas eran las que utilizaban *P4'* en el caso de *R1* y *P3'* en el caso de *R2*, así que descartamos los candidatos que generaban estos punto, quedándonos finalmente con 4 candidatos para cada recta proyectiva, es decir, 8 en total.

4.4.1. Cálculo del coste optimizando en λ

En cada distancia elegida (λ) dentro de la recta proyectiva que estemos recorriendo, debemos calcular un coste para cada candidato con el fin de poder hacer una comparación entre ellos y elegir cuál es el mejor.

Para calcular el coste, hay diversas propiedades de la portería que pueden utilizarse. Después de realizar varias pruebas, la función de coste que mejores resultados nos ha dado ha sido la que usaba únicamente los ángulos entre postes, larguero y diagonales de la portería, haciendo las siguientes comprobaciones:

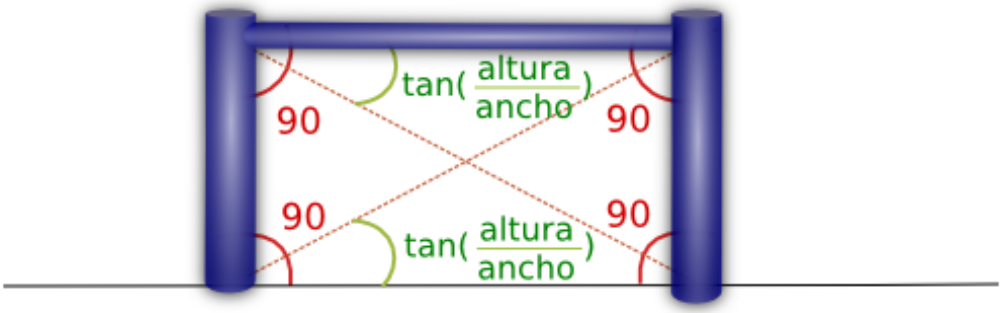
- Poste derecho (*P1-P3*) y larguero (*P3-P4*): 90 grados.
- Poste izquierdo (*P2-P4*) y larguero (*P3-P4*): 90 grados.
- Diagonal1 (*P1-P4*) y larguero (*P3-P4*): tan(altura/ancho).
- Poste derecho (*P1-P3*) y suelo (*P1-P2*): 90 grados.
- Poste izquierdo (*P2-P4*) y suelo (*P1-P2*): 90 grados.
- Diagonal2 (*P2-P3*) y suelo (*P1-P2*): tan(altura/ancho).

Todos estos ángulos pueden verse gráficamente en la figura 4.31.

Conociendo los ángulos anteriores comprobamos los ángulos de nuestros puntos calculados, siendo el coste, el sumatorio de las diferencias de cada ángulo (4.14).

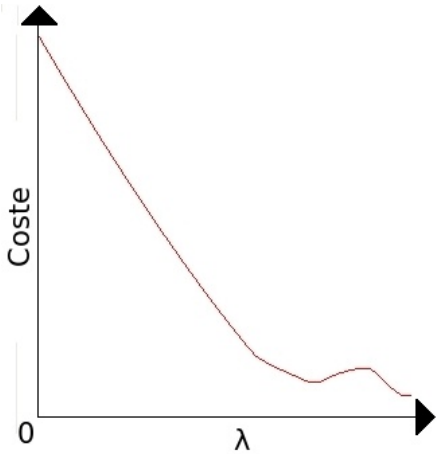
$$\sum_{0 \leq i \leq 5} |\alpha_i - calculado_i| \tag{4.14}$$

Podemos ver un ejemplo del coste obtenido por cada "ganador" de cada iteración en la imagen 4.32.



(a)

Figura 4.31: Ángulos de la portería.



(a)

Figura 4.32: Coste obtenido para cada λ.

4.4.2. Transformación de coordenadas relativas a absolutas

La posición de la portería obtenida con el algoritmo de las esferas, es una posición de la portería relativa respecto al robot, sin embargo, para poder localizar al robot en el campo, necesitamos conocer la posición absoluta respecto al sistema de coordenadas del campo.

Por lo tanto, necesitamos invertir la información, partiendo de un sistema de referencia absoluto donde se conoce la posición de las porterías, tenemos que conocer la posición de la cámara usando la posición relativa de la portería que hemos calculado.

Para la transformación de coordenadas relativas a absolutas existen diversos algoritmos, algunos de ellos comentados en [Larusso *et al.*, 1995]. Nosotros hemos decidido implementar el algoritmo SVD (*Singular Value Decomposition*), puesto que era el que mejor se adaptaba a nuestras necesidades en tiempo de cómputo y fiabilidad de la transformación.

Descripción del algoritmo SVD

Suponiendo que existe un conjunto de puntos $Pabs_i$ (puntos absolutos) y $Prel_i$ (puntos relativos), con $i = 1..N$, que están relacionados de la forma:

$$Prel_i = R * Pabs_i + T + V \tag{4.15}$$

donde R es una matriz de rotación de 3x3, T es un vector de traslación de 3 dimensiones y V_i un vector de ruido. Resolver la ecuación para los $[\hat{R}, \hat{T}]$ óptimos que transforman $Pabs_i$ en $Prel_i$, normalmente requiere minimizar el error mediante mínimos cuadrados, utilizando la ecuación 4.16:

$$(\sum)^2 = \sum_{i=1}^{N} ||Prel_i + R * Pabs_i + \hat{T}||^2 \tag{4.16}$$

Teniendo en cuenta que los conjuntos de puntos tendrán el mismo centroide, primero calculamos los centroides de ambos conjuntos de puntos con las ecuaciones 4.17 y 4.18.

$$PCentRel = \frac{\sum_{i=1}^{N} Prel_i}{N} \tag{4.17}$$

$$PAbsRel = \frac{\sum_{i=1}^{N} Pabs_i}{N} \tag{4.18}$$

Una vez calculados los centroides podemos calcular H mediante 4.19:

$$H = \sum_{i=1}^{N} (Prel_i - PCentRel) * (Pabs_i - PCentAbs)^T \tag{4.19}$$

Al realizar la descomposición en valores singulares de H (4.20), podemos calcular la matriz de rotación $\hat{R}$ con la ecuación 4.21.

$$H = U\Lambda V^T \tag{4.20}$$

$$\hat{R} = V * U^T \tag{4.21}$$

El vector de traslación, puede ser calculado ahora despejando de la ecuación 4.22.

$$\hat{T} = PCentRel - \hat{R} * PCentAbs \tag{4.22}$$

Cuando utilizamos como conjuntos de entrada datos coplanares o en presencia de mucho ruido se produce un caso particular, puede que el determinante de $\hat{R}$ sea -1, en este caso, los cálculos anteriores llevarían al punto espejo en vez de al de rotación. Esto se puede corregir estableciendo:

$$\hat{R} = V' * U^T \tag{4.23}$$

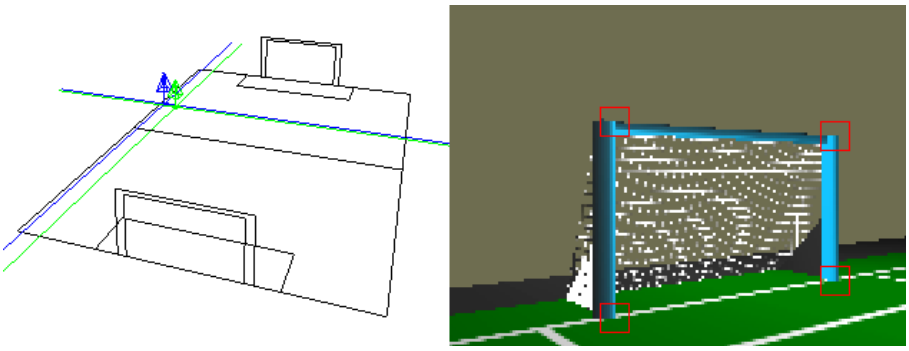
donde $V' = [v_1, v_2, -v_3]$, es decir, cambiamos de signo a la tercera columna de V'.

Con esto el algoritmo funciona para cualquier conjunto de datos que tome como entrada, en nuestro caso, hemos utilizado siempre 4 puntos, que corresponden a cada esquina de la portería.

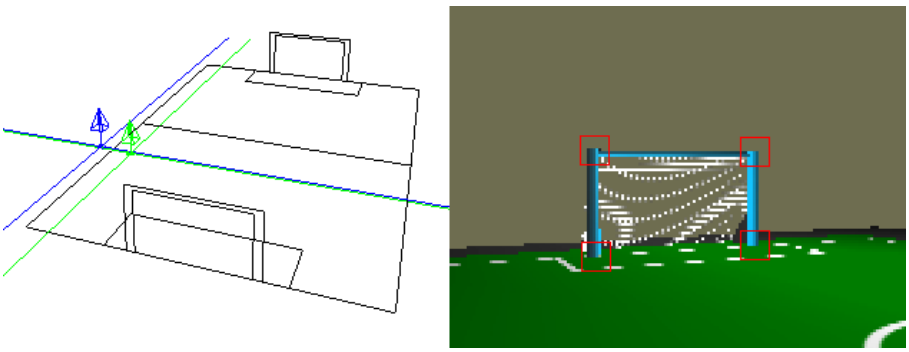
4.4.3. Experimentos

En el caso del algoritmo de las esferas, hemos escrito un total de 430 líneas de código y el tiempo de ejecución medio ha sido de 20 milisegundos, donde al igual que en el algoritmo anterior, la mayor parte (17 milisegundos) se corresponden con la detección de la portería, y tan solo 3 milisegundos son del propio algoritmo.

Para poder analizar el algoritmo al igual que lo hicimos con el de los toros, vamos a mostrar las mismas imágenes que en el caso anterior, para después poder comparar ambos algoritmos con más rigor. Primero podemos ver las capturas tomadas para el simulador Webots en 4.33 y 4.34.



(a)



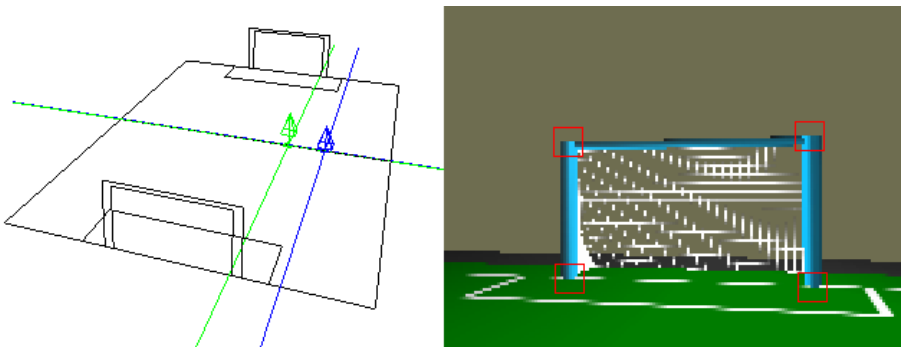
(b)

Figura 4.33: Localización instantánea usando esferas en Webots.

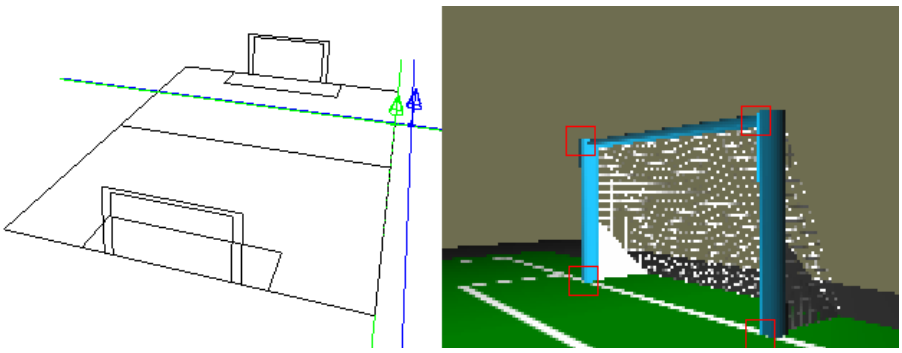
Podemos ver cómo en ocasiones la precisión no es demasiado buena, esto es debido a que la función de coste que utilizamos da mejores resultados para el candidato que se ha elegido, a pesar de no ser el que se encuentra a menor distancia del punto real, debido a las propiedades geométricas de los puntos de dicho candidato.

En este caso, el error medio medido para las 4 imágenes ha sido de 34.6 cm, aunque hay que tener en cuenta que este algoritmo también tiene como variable la altura (z), al

contrario de lo que ocurre con el algoritmo de los toros.



(a)



(b)

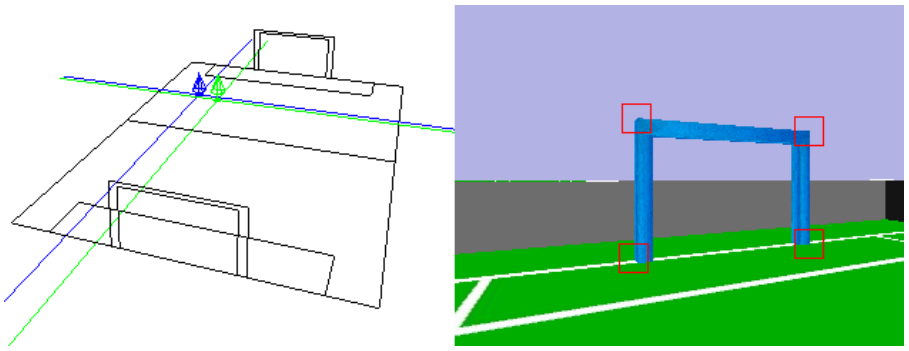
Figura 4.34: Localización instantánea usando esferas en Webots.

Al igual que el algoritmo anterior, al ver la portería amarilla el algoritmo actúa de la misma forma y sólo es necesario modificar las coordenadas x e y a la hora de guardar la posición calculada.

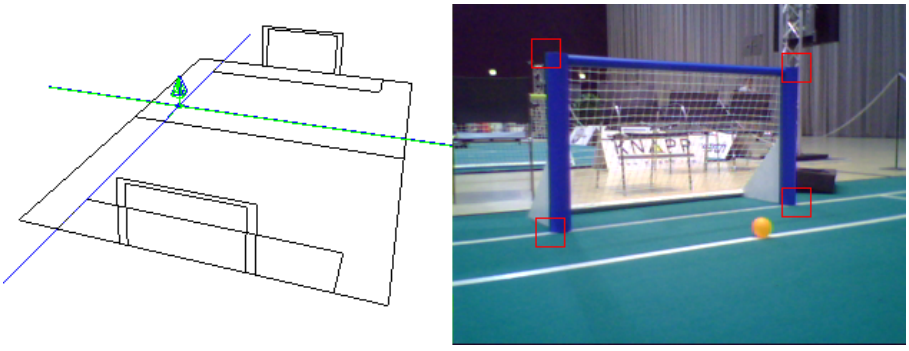
Cuando utilizamos el simulador Gazebo o las imágenes reales los resultados son similares a lo sucedido con el simulador Webots, como podemos ver en la figura 4.35.

El algoritmo es bastante sensible a los píxeles dados como entrada, por ello, si al igual que en el algoritmo anterior detectamos mal la portería, se producirá un error en la localización, e incluso algunas veces puede que la localización obtenida cambie totalmente de lugar, ya que la función de coste puede elegir una posición totalmente distinta con sólo una leve modificación, como puede verse en 4.36.

El comportamiento del algoritmo varía en función de la zona del campo en la que nos encontremos, siendo más preciso cuando nos encontramos en los laterales que en la zona central. Además, al igual que sucedía con el algoritmo de los toros, la precisión también empeora cuando nos alejamos de la portería.

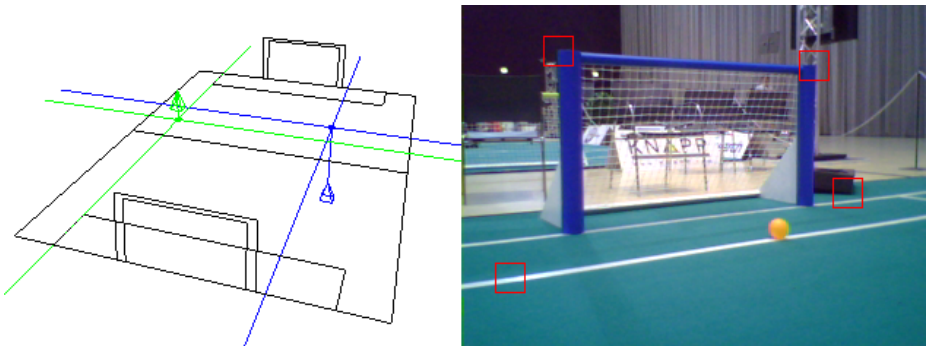


(a)



(b)

Figura 4.35: Imagen utilizando el simulador Gazebo (a) e imagen real tomada del Nao (b).



(a)

Figura 4.36: Error en la localización instantánea usando esferas.

Como ya hemos visto, la sensibilidad del algoritmo es grande, pudiendo obtener resultados no esperados en el caso de no detectar correctamente la portería, por lo que hay que tener especial cuidado con los parámetros de los filtros de color y la transformada de Hough.

Por otra parte, este algoritmo funciona sin necesidad de conocer la altura del robot, pudiendo adaptarse a cambios de altura durante la ejecución y ofreciéndonos una gran versatilidad.

Otras funciones de coste para las esferas

Antes de seleccionar como función de coste el método descrito en 4.4.1, donde se tienen en cuenta los distintos ángulos que forman la portería, hemos probado otras funciones de coste como las que vamos a describir a continuación.

La primera función de coste que utilizamos fue la distancia entre los puntos calculados, de modo que una vez obtenidos los puntos candidatos *P2*, *P3* y *P4* tras intersecar cada recta proyectiva con la esfera generada, comparábamos las distancias de estos puntos entre sí con las que deberían tener en la realidad, lo que era posible ya que conocíamos las dimensiones reales de la portería.

Los resultados obtenidos con este método no fueron tan satisfactorios como con el método finalmente seleccionado, puesto que se producían más errores al evaluar los candidatos.

Otra de las alternativas que manejamos buscando más precisión fue la de mezclar el método anterior con el de los ángulos, teniendo en cuenta tanto los ángulos entre los distintos segmentos de la portería como las distancias entre los puntos.

Sin embargo, las mejoras no eran apreciables, y la pérdida de eficiencia en tiempo que se producía al tener que realizar más cálculos para cada uno de los candidatos, era demasiado grande en comparación con la mejora obtenida.

Por ello, y teniendo en cuenta tanto la precisión como la eficiencia, decidimos utilizar únicamente los ángulos entre los distintos segmentos de la portería para evaluar a cada uno de los candidatos, convirtiéndose este método en el seleccionado finalmente como función de coste para nuestro algoritmo.

4.5. Comparativa entre los algoritmos de localización 3D instantánea

Con el análisis hecho para los dos algoritmos pueden sacarse algunas conclusiones respecto a la conveniencia de su utilización.

El primer aspecto a tener en cuenta es la complejidad de los algoritmos. Como puede verse, el algoritmo de los toros da como resultado una localización obtenida analíticamente y tan solo ha ocupado unas 100 líneas de código, por el contrario, el algoritmo de las esferas depende de una función de coste cuya precisión no es siempre la adecuada, ocupando además 4 veces más líneas de código.

Además ambos algoritmos tienen un tiempo de ejecución similar, aunque el primero de los algoritmos es ligeramente más rápido.

En cuanto a la precisión, el algoritmo de los toros suele tener más precisión a la hora de localizarnos, además de que cuando falla en la localización, lo hace en menor proporción que el de las esferas.

Por todo lo anterior, podríamos concluir que el algoritmo de los toros tiene un mejor funcionamiento y una fiabilidad mayor, sin embargo, hay que tener en cuenta que los toros fueron calculados suponiendo una altura constante del robot, ya que en nuestro caso era conocida.

Por su parte, el algoritmo de las esferas no depende de una altura dada, por lo que se podría adaptar a otros casos donde no se conociese la altura del robot utilizado o donde ésta variase a lo largo del tiempo.

Capítulo 5

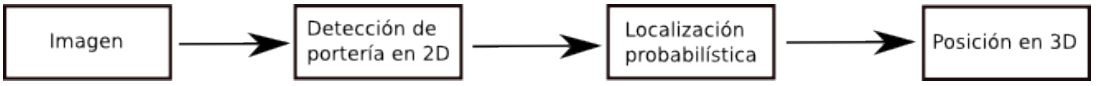
Localización 3D probabilística

Durante la descripción de los algoritmos de localización instantánea, vimos como en los casos en los que la detección de la portería no era suficientemente buena se podrían producir errores en la localización. Estos errores en la detección podían ser debidos al ruido de la imagen, oclusiones, fallos en los filtros de color, etc., y en ocasiones no iban más allá de una o dos observaciones independientes.

Estos errores en observaciones concretas, podrían subsanarse utilizando un modelo de probabilidad, en el cual los pequeños errores quedarían ocultos frente a la probabilidad acumulada del resto de observaciones. Además este tipo de métodos ya han sido utilizados con éxito en otros proyectos, por lo que su funcionamiento está demostrado.

Por todo ello, hemos decidido adaptar los algoritmos comentados en el capítulo 4 para poder hacer uso de la probabilidad, y así poder calcular la posición del robot mediante localización probabilística.

El proceso a seguir es similar al que describíamos en el capítulo 4, como puede verse en la figura 5.1. Al igual que en la localización instantánea, se analiza la imagen recibida en busca de las 4 esquinas que definen la portería y se determina qué portería estamos viendo. Esta información se transmite al algoritmo de localización probabilística, que nos devuelve una posición en 3D calculada mediante probabilidad.



(a)

Figura 5.1: Proceso seguido en la localización probabilística.

5.1. Fundamentos de la localización probabilística

Los algoritmos probabilísticos han sido siempre de gran utilidad para resolver problemas de localización como el que estamos abordando.

Este tipo de localización utiliza un mapa de probabilidad, en el que cada posición (x, y, z) tiene una probabilidad asociada. Esta probabilidad indica en qué grado puede encontrarse el robot en esa posición, e irá modificándose con el paso del tiempo según la información obtenida.

Se utiliza un modelo de observación que utiliza la información recibida desde los distintos sensores del robot para evaluar la verosimilitud de que la información recibida se haya realizado desde una determinada posición. Los sensores a utilizar, pueden ir desde sensores de ultrasonido o sensores láser, que nos presentan la información en forma de distancias a objetos, hasta cámaras, que nos presentan una imagen en forma de matriz numérica.

Para calcular la probabilidad también pueden utilizarse modelos de movilidad, en los que la información que se obtiene es lo que se ha movido el robot desde la última observación, esto puede conseguirse mediante dispositivos como el GPS o utilizando la odometría del robot.

La acumulación de probabilidades se realiza utilizando la fusión de Bayes, en la que la probabilidad calculada en la última observación se fusiona con la probabilidad calculada anteriormente, haciendo que se modifique esta probabilidad acumulada según vayan apareciendo nuevas observaciones.

5.2. Localización visual en la RoboCup

En nuestro caso, el mapa de probabilidad que se ha utilizado es un cubo de probabilidad en 3D que abarca todas las posibles posiciones del campo de la RoboCup.

Cada celda del cubo de probabilidad representa un cubo de 50 milímetros de ancho en cada una de las 3 dimensiones, de modo que en total, el cubo abarca 6050 milímetros de largo, 4050 milímetros de ancho y 1050 de alto. El alto y ancho coinciden con las dimensiones del campo de la RoboCup para la plataforma estándar, mientras que la altura es una aproximación teniendo en cuenta la altura real del robot Nao (500 milímetros).

Para conocer la probabilidad de cada celda, se han utilizado tres modelos de observación distintos, dos de ellos utilizando las porterías, y otro utilizando las líneas del campo, que

serán descritos más adelante.

En el caso de robot móviles, como es nuestro caso, para calcular la nueva probabilidad podemos tener en cuenta lo que se ha desplazado el robot desde la última observación, haciendo uso de la odometría como ya comentamos en el capítulo 1.4. Sin embargo, al ser nuestro robot un robot con patas, el cálculo del desplazamiento realizado es muy complejo y por lo tanto no se utiliza en la práctica.

Como ya hemos comentado, para la actualización de cada celda, hay que tener en cuenta tanto la probabilidad ya calculada anteriormente, a la que llamaremos probabilidad acumulada, como la probabilidad de la última observación, a la que haremos referencia como probabilidad instantánea.

Una vez calculada tanto la probabilidad acumulada, como la instantánea, la posición del robot vendrá dada por la celda cuya probabilidad acumulada sea mayor. En el caso de que varias celdas tengan la misma probabilidad acumulada, se seleccionará la celda que tenga mayor probabilidad instantánea dentro de las que obtuvieron mayor acumulada.

5.2.1. Acumulación de observaciones parciales

En principio, todas las celdas del cubo de probabilidad tienen una probabilidad del 0.5 cuando se inicia la ejecución del programa, ya que no disponemos de ninguna información a priori.

A partir de aquí, se calcula con cada nueva observación un nuevo cubo de probabilidad en el que únicamente se guarda la probabilidad instantánea con alguno de los modelos de observación que veremos más adelante.

El valor de cada celda de este nuevo cubo de probabilidad instantáneo, se limita entre dos valores con el fin de evitar el escalado demasiado rápido de la probabilidad acumulada y para guardar una cierta prudencia a la hora de acumular las observaciones. Después de realizar diversas pruebas en este sentido, hemos decidido establecer un rango que va desde 0.4 hasta 0.6 y donde cualquier valor superior o inferior es truncado a esos valores.

Una vez que tenemos los valores del cubo de probabilidad instantáneo, debemos recorrer el cubo de probabilidad acumulado para determinar el valor de cada celda, teniendo en cuenta el valor de esa celda en ambos cubos de probabilidad.

Para acumular las observaciones hemos utilizado un modelo probabilístico de Bayes, utilizando la ecuación 5.1, donde p será el nuevo valor del cubo de probabilidad, p_i es la probabilidad instantánea que se acaba de calcular, y p_a es la probabilidad acumulada.

$$\ln p = \ln \frac{p_i}{1 - p_i} + \ln \frac{p_a}{1 - p_a} \tag{5.1}$$

Al igual que hicimos con el cubo de probabilidad instantáneo, hemos limitado la probabilidad acumulada dentro de un rango de valores para evitar los valores extremos, es decir, 0 y 1. En este caso el rango de probabilidad se ha fijado entre 0.1 y 0.9.

Gracias a este nuevo rango, la memoria temporal se recorta, permitiendo que el algoritmo se adapte rápidamente a cambios en la probabilidad instantánea.

Si no realizásemos este rango, en el caso de que nos encontrásemos en una celda con probabilidad p_t, y obtuviésemos un número N de observaciones positivas para esa celda, para volver a tener la misma probabilidad p_t necesitaríamos el mismo número N de observaciones negativas, haciendo que el algoritmo cambiase muy lentamente de opinión. Esto se produce porque la fusión Bayesiana cumple la propiedad asociativa.

Además de acumular las observaciones por el método que acabamos de describir, hemos desarrollado otra técnicas para mejorar el cálculo de la probabilidad acumulada en cada celda, como veremos en la siguiente sección.

5.2.2. Observaciones independientes

A la hora de calcular la probabilidad es posible que el robot no se mueva durante un cierto periodo de tiempo, y por tanto siempre obtenga la misma imagen de la cámara. Si se actualiza el cubo de probabilidad acumulada cada pocos milisegundos en el caso anterior, incluiremos varias veces la misma observación en la probabilidad, haciendo que se alcancen resultados poco fiables.

Con el fin de evitar lo anterior, hemos generado una función que detecta cuándo la imagen observada ha cambiado respecto a la última tenida en cuenta, de modo que sólo se actualiza la probabilidad acumulada cuando haya cambiado la imagen observada.

Esto además tiene como consecuencia que el algoritmo de localización probabilística no se ejecute en todas las iteraciones, haciendo que disminuya el tiempo de ejecución del programa.

Otra de las ventajas obtenidas al utilizar esta funcionalidad, es que evita que la probabilidad acumulada escale demasiado rápido hacia los máximos permitidos. No obstante, en el momento en el que el robot se mueva, la observación cambiará con regularidad, lo que nos asegura que la probabilidad acumulada se actualizará con la suficiente frecuencia como para para obtener una localización precisa en cada momento.

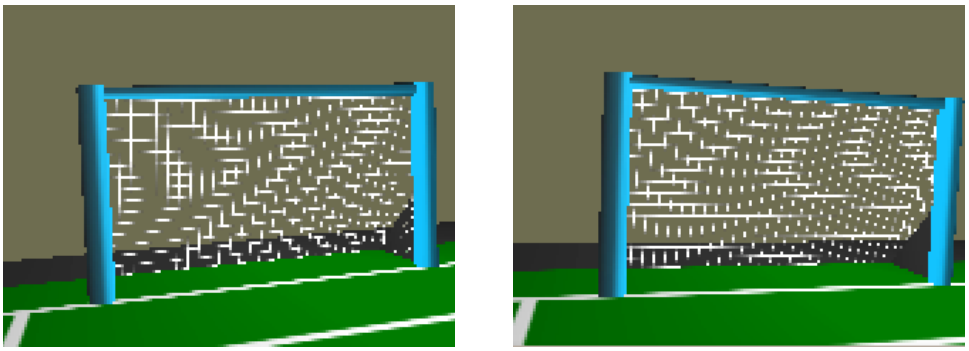
Para realizar este algoritmo, podríamos haber tenido en cuenta la distancia recorrida por el robot usando la odometría, sin embargo, como ya dijimos, es difícil calcular la movilidad del robot por medio de la odometría para robots con patas.

Por ello, la técnica elegida para determinar si nos encontramos ante una nueva observación, consiste en comprobar si los píxeles calculados en la detección de las porterías han cambiado significativamente respecto a la observación anterior.

Una vez probado el algoritmo, hemos visto que el movimiento de la cámara al moverse el robot es muy alto, lo que hace que los píxeles de las esquinas de la portería cambien con mucha facilidad.

Esto puede verse en la figura 5.2, donde se muestran dos imágenes tomadas con tan solo medio segundo de diferencia, y donde puede comprobarse como las esquinas de la portería han cambiado de posición de manera notable, a pesar de haber pasado tan poco tiempo entre ambas imágenes.

Así que, para no ser tan sensibles a cambios de imagen pequeños, decidimos fijar un umbral de 5 píxeles para que los píxeles detectados se tuviesen que desplazar lo suficiente como para considerar que la observación había cambiado.



(a) (b)

Figura 5.2: Imágenes tomadas con 0.5 segundos de diferencia al moverse el robot.

5.3. Modelo de observación con porterías y ángulos

El objetivo de los modelos de probabilidad que vamos a ver, es el de calcular los cubos de probabilidad instantáneos que permitirán calcular la probabilidad acumulada como acabamos de ver en la sección anterior.

Este modelo de observación recibe como entrada los 4 píxeles detectados con el algoritmo de detección de porterías visto en el capítulo 4, que representan las 4 esquinas de la portería, y también recibe un parámetro para indicar qué portería estamos viendo, la azul o la amarilla, lo que se tendrá en cuenta a la hora de calcular el cubo de probabilidad.

El primer modelo de observación que hemos desarrollado está basado en el algoritmo de localización instantánea usando toros visto en el capítulo 4.3.

Al igual que en ese algoritmo, se utiliza la librería Progeo para calcular las rectas proyectivas de los 4 píxeles dados como entrada. Una vez obtenidas las rectas proyectivas, se toma cada poste por separado, que en este caso corresponden con las rectas proyectivas de *Pix1-Pix3* y las de *Pix2-Pix3*.

En cada poste, se calcula el ángulo α real que forman las dos rectas proyectivas, despejando este valor de la ecuación del producto escalar que podemos ver en 5.2.

$$uv = |u||v|cos(\alpha) \tag{5.2}$$

En el algoritmo de los toros de 4.3, este α calculado se utilizaba para obtener el toro de cada poste. Sin embargo, para este modelo de observación, no es necesario realizar más cálculos geométricos, sino que este ángulo nos sirve para el cálculo de la probabilidad como veremos a continuación.

Una vez obtenido el ángulo que forma cada poste en la imagen, debemos actualizar todas las celdas del cubo de probabilidad instantáneo con este valor. Para ello, recorremos todo el cubo de probabilidad y calculamos en cada celda el ángulo que deberíamos obtener si nos encontrásemos en esa posición, es decir, obtenemos un α teórico.

Para calcular este α teórico mediante la ecuación 5.2, necesitamos obtener dos vectores a partir de 3 puntos conocidos. Dos de esos puntos son las posiciones en coordenadas absolutas de los extremos del poste (teniendo en cuenta si vemos la portería azul o la amarilla), y el tercer punto puede obtenerse calculando la posición real en 3D que se corresponde con la celda en la que estamos, lo que es muy fácil puesto que conocemos la equivalencia de cada celda con la realidad y el número de celda en el que nos encontramos dentro del cubo de probabilidad.

Una vez que tenemos los 3 puntos, el extremo superior Ps, el extremo inferior Pi y nuestra posición actual Pa, obtenemos los vectores formados por Ps-Pa y Pi-Pa, que se obtienen con las ecuaciones 5.3. Tras calcular estos dos vectores, ya podemos obtener el ángulo α teórico en cada celda igual que lo hicimos para el α real, con la ecuación 5.2

$$\begin{aligned} \overrightarrow{PsPa} &= (Pa_x - Ps_x, Pa_y - Ps_y, Pa_z - Ps_z) \\ \overrightarrow{PiPa} &= (Pa_x - Pi_x, Pa_y - Pi_y, Pa_z - Pi_z) \end{aligned} \tag{5.3}$$

Para calcular la probabilidad instantánea de cada celda, nos interesa conocer la diferencia entre estos ángulos en grados, para lo que utilizamos la ecuación 5.4, donde α_r es el ángulo α real y α_t es el ángulo α teórico, estando ambos ángulos en radianes.

$$diff = \frac{|\alpha_r - \alpha_t| * 180}{\pi} \tag{5.4}$$

La probabilidad para cada poste se calcula teniendo en cuenta esta diferencia, mediante la ecuación 5.5.

$$p = \begin{cases} 0 & \text{si } diff \geq \text{ 1 grado} \\ 1 - diff & \text{si } diff < \text{ 1 grado} \end{cases} \tag{5.5}$$

Al realizar la misma operación para los dos postes, obtendremos dos probabilidades en cada celda, una para el poste derecho y otra para el izquierdo, calculando la probabilidad instantánea final de la celda mediante la ecuación ya vista en 5.1.

5.3.1. Experimentos

Al igual que en la localización instantánea, hemos realizado una serie de experimentos con el fin de mejorar y depurar nuestros algoritmos.

En el algoritmo de probabilidad desarrollado hemos añadido la opción de poder probar cada modelo de observación en 2 dimensiones, teniendo en cuenta la altura del robot, o en 3 dimensiones, permitiendo una mayor generalidad a la solución. De este modo podremos comparar cada uno de los modelos de observación desarrollados en igualdad de condiciones con los algoritmos instantáneos en los que se basan.

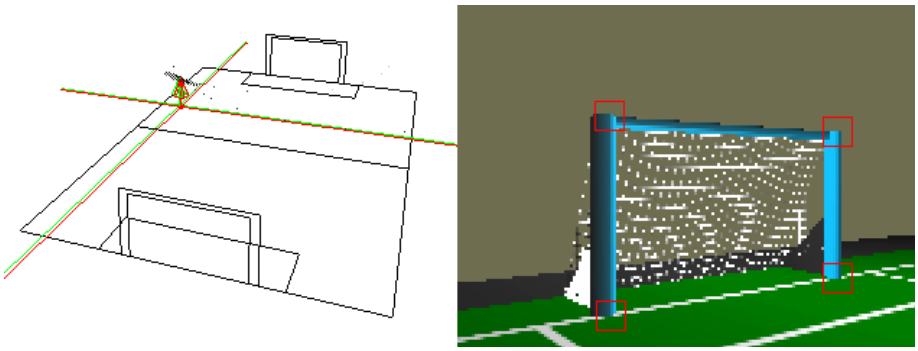
El modelo de observación que acabamos de describir ha ocupado unas 140 líneas de código, y al utilizarlo, el algoritmo de probabilidad tiene unos tiempos de ejecución de 54

milisegundos en la versión 2D y de 290 milisegundos en la versión 3D. Como podemos ver, la versión en 3D requiere muchos más recursos, por lo que por temas de eficiencia vamos a suponer conocida la altura del robot y vamos a utilizar la versión en 2 dimensiones.

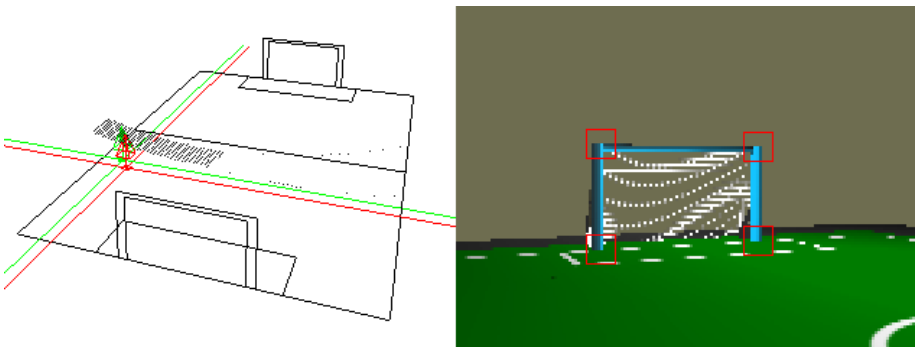
Como ya hicimos con los algoritmos instantáneos, vamos a mostrar varias imágenes en el simulador Webots para comprobar el correcto funcionamiento del algoritmo, utilizando la versión en 2D, lo cual podemos ver en las imágenes mostradas en 5.3 y 5.4.

En este caso, en las imágenes pueden apreciarse varios elementos, la flecha verde, que al igual que en los algoritmos instantáneos representaba la posición real, la flecha roja, que representa la posición calculada con el algoritmo probabilístico, y distintos puntos en escala de grises repartidos por la imagen, que indican las zonas donde la probabilidad acumulada es más alta.

Para los 4 casos mostrados en el simulador Webots, el error medio ha sido de unos 13.3 centímetros, valor bastante similar al obtenido para la localización instantánea. Otro factor a tener en cuenta, es la forma en la que aumenta la zona con probabilidad más alta a medida que nos alejamos de la portería, aumentando así la incertidumbre con la distancia.

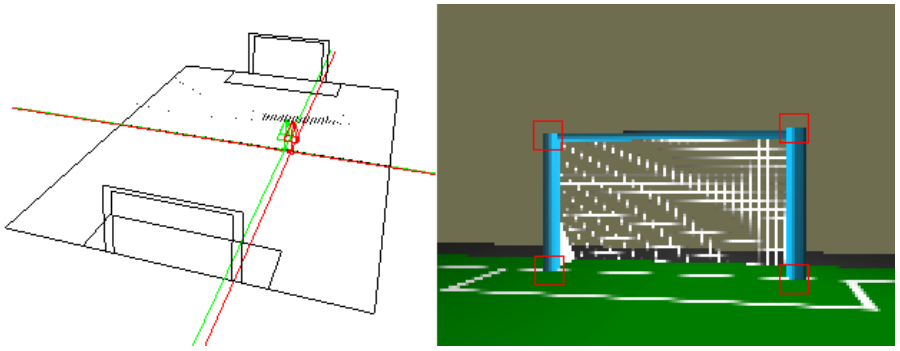


(a)

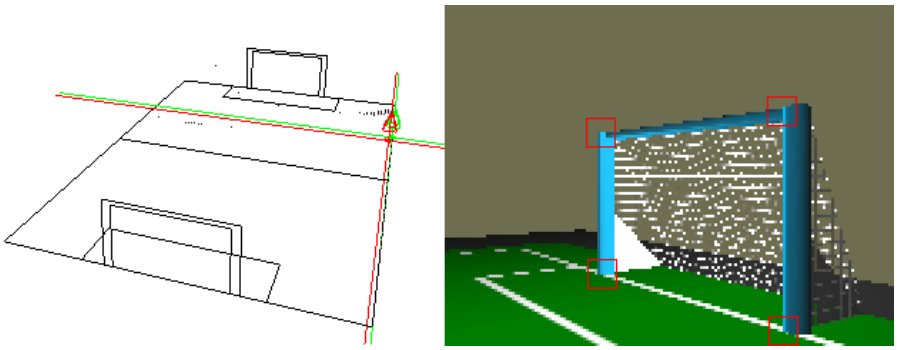


(b)

Figura 5.3: Localización probabilística usando ángulos en Webots.



(a)



(b)

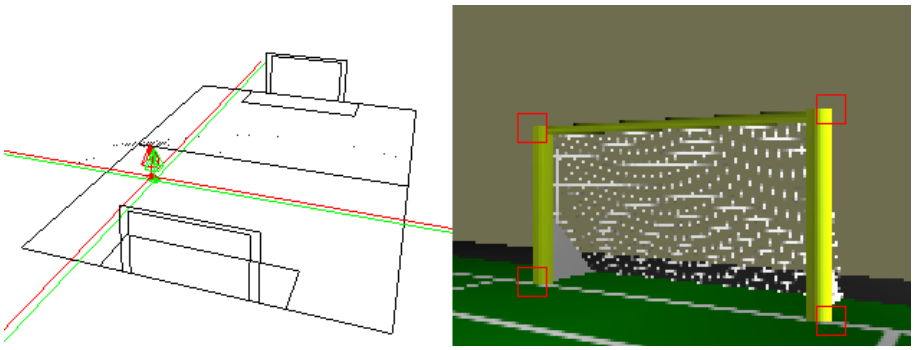
Figura 5.4: Localización probabilística usando ángulos en Webots.

Si la portería detectada es la amarilla, en los cálculos realizados desde cada α teórico de cada celda, se habrán tenido en cuenta las posiciones absolutas de los postes de la portería amarilla, y por lo tanto los cálculos realizados no necesitan ser invertidos como sucedía en los algoritmos instantáneos. Podemos ver el funcionamiento con la portería amarilla en 5.5.

En el caso de que utilicemos el simulador Gazebo (5.6(a)) o las imágenes reales del Nao (5.6(b)), los resultados también son similares a los obtenidos con Webots.

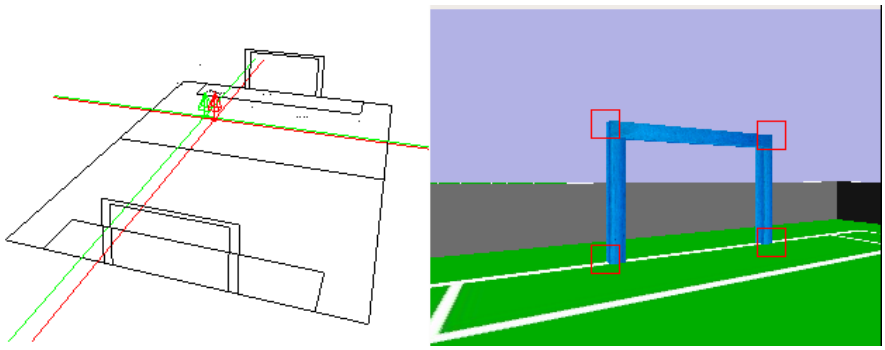
Como ya hemos dicho, las imágenes mostradas hasta ahora se han realizado sobre un cubo de probabilidad en 2D, suponiendo la altura del robot fija. El modelo también permite la utilización del cubo de probabilidad completo, calculando la posición del robot en 3D y con resultados satisfactorios, como puede verse en 5.7. Aunque como podemos observar, la zona marcada con alta probabilidad es muy grande y los recursos consumidos son mucho mayores.

Este modelo de observación tiene unas propiedades similares a las que tenía el algoritmo instantáneo usando toros, puesto que ambos se basan en los ángulos que forman los postes, por ello, la precisión obtenida con el modelo de observación en este caso es muy similar en

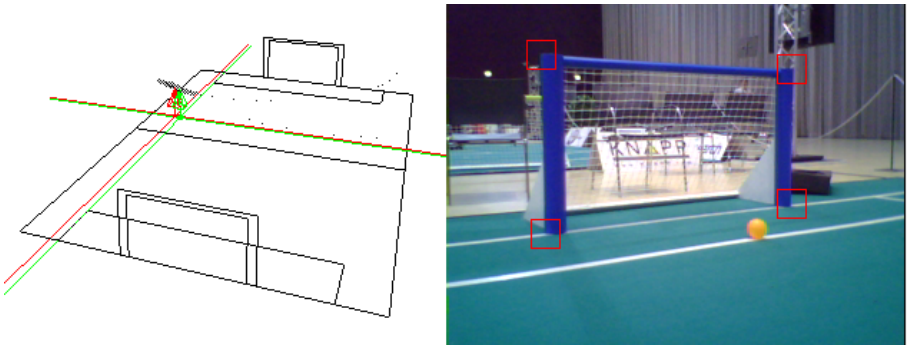


(a)

Figura 5.5: Localización probabilística usando ángulos con portería amarilla.

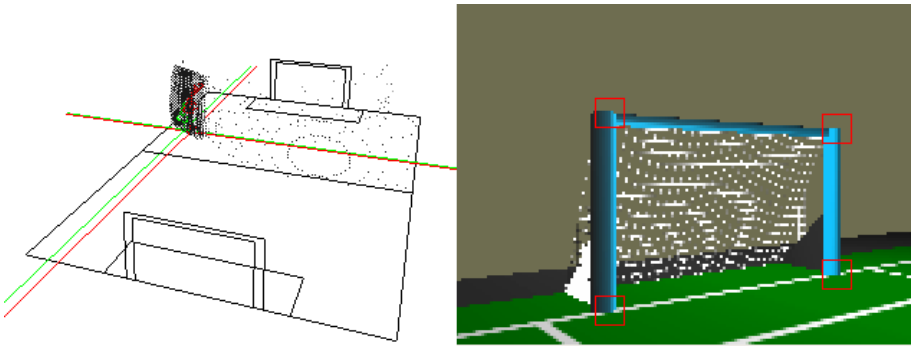


(a)



(b)

Figura 5.6: Imagen utilizando el simulador Gazebo (a) e imagen real tomada del Nao (b).



(a)

Figura 5.7: Localización probabilística usando ángulos y el cubo en 3D.

todas las zonas del campo, perdiendo precisión a medida que nos alejamos de la portería y aumentando además la incertidumbre.

Cabe destacar, que al contrario de lo que ocurría en la localización instantánea, un error puntual en la observación no hace que el resultado sea incorrecto, puesto que la posición calculada depende de varias observaciones. Por tanto, para que se cometiese un error en la localización debería producirse un error en la detección de la portería a lo largo de varias observaciones continuas.

5.4. Modelo de observación con porterías y esferas

Al igual que en el modelo de observación que acabamos de ver, el objetivo que buscamos con este nuevo modelo de observación es el de calcular el cubo de probabilidad instantáneo a partir de los datos de entrada recibidos, es decir, las 4 esquinas de la portería y el parámetro que indica qué portería estamos viendo.

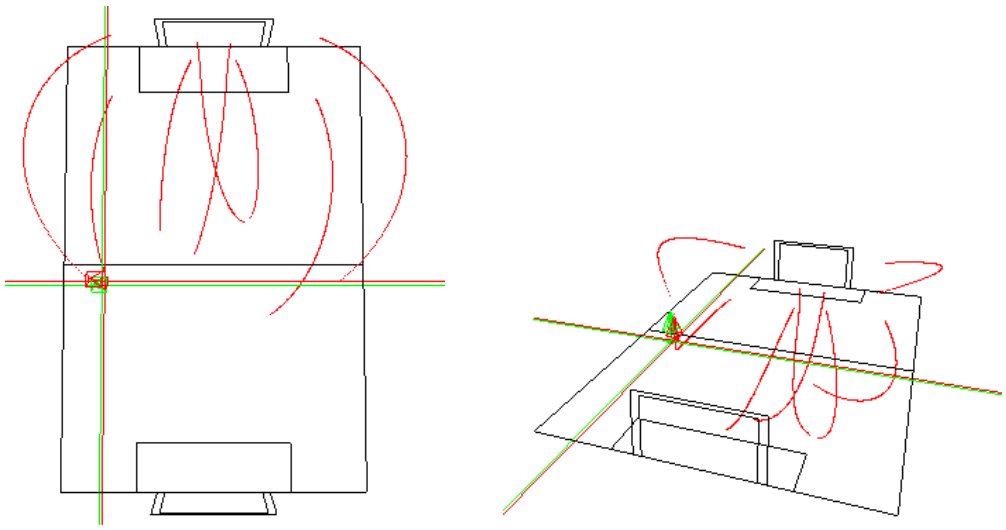
Este segundo modelo de observación desarrollado, está basado en el algoritmo de localización instantánea usando esferas que pudimos ver en 4.4.

El algoritmo es muy similar al utilizado en la localización instantánea, y su funcionamiento básico es el mismo, recorriendo las rectas proyectivas de *Pix1* (*R1*) y de *Pix2* (*R2*) para obtener 8 candidatos para cada λ, donde 4 son de ellos se obtendrán con *R1* y los otros 4 con *R2*.

Todos los candidatos resultantes son evaluados mediante una función de coste, pero a diferencia del algoritmo instantáneo, no nos quedamos con un ganador para cada λ, sino que tenemos en cuenta todos los candidatos.

Es decir, para cada λ, obtendremos 8 candidatos, y de cada candidato obtendremos una probabilidad que será utilizada en el cubo de probabilidad instantáneo.

En la imagen 5.8 podemos ver un ejemplo donde se muestran en rojo todos los candidatos para una determinada observación.



(a) (b)

Figura 5.8: Puntos candidatos para la localización en 2D (a) y 3D (b).

Para obtener los candidatos mostrados en la imagen, hemos tenido que realizar una transformación a coordenadas absolutas utilizando para ello el método que vimos en 4.4.2, ya que la posición que se obtiene en principio para cada candidato es una posición relativa respecto a la cámara.

Una vez que conocemos cuál es la posición en coordenadas absolutas, debemos tener en cuenta qué portería estamos viendo. Si la portería que vemos es la azul, la posición calculada será la correcta, sin embargo, si estamos viendo la portería amarilla, necesitaremos modificar las coordenadas X (profundidad) e Y (ancho), para situarnos justo en la otra esquina del campo, al igual que sucedía con los algoritmos de localización instantánea.

Para el cálculo del cubo de probabilidad instantáneo, no se recorre cada celda como en el modelo de observación anterior, sino que se inicializan todas las celdas del cubo con probabilidad 0 y sólo se modifican las celdas en las que hayamos encontrado un candidato.

A la hora de calcular la probabilidad de un candidato, obtenemos la celda adecuada dentro del cubo de probabilidad instantáneo, mediante una función muy simple que nos indica la celda que se corresponde con cada posición real.

La probabilidad instantánea de la celda seleccionada se calcula mediante la ecuación 5.6, en la que F_c es el valor obtenido en la función de coste para el candidato que estemos utilizando.

$$p(x, y, z) = 1 - \frac{\ln\left(\frac{F_c}{2} + 1\right)}{\ln 100} \tag{5.6}$$

Esta ecuación se ha obtenido teniendo en cuenta que en el mejor de los casos, la función de coste devuelve un resultado de 0, que se corresponde con una probabilidad de 1, y en el peor de los casos, la función dará como resultado un valor mayor o igual a 198, que se corresponde con una probabilidad de 0. De este modo, la probabilidad de 0.5 se obtendrá para un valor de la función de coste de 18.

Por último, para evitar que un candidato sólo modifique una única celda, lo cual puede ser insuficiente al acumular probabilidades, hemos propagado esta probabilidad entre las celdas adyacentes, teniendo en cuenta además, que en el caso de que varios candidatos modifiquen una misma celda, nos quedaremos con la probabilidad que sea mayor.

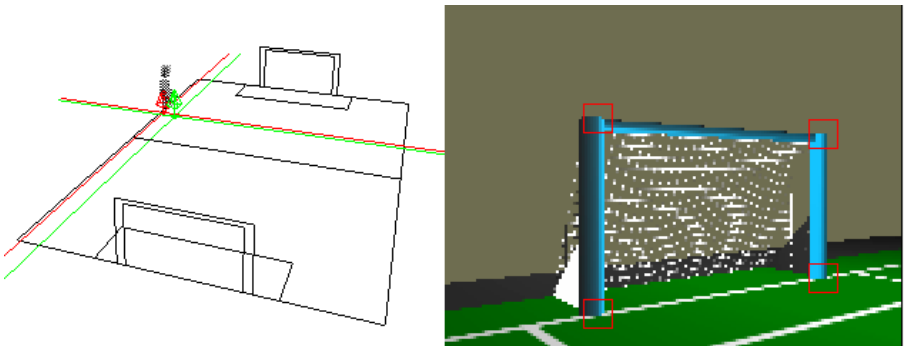
5.4.1. Experimentos

Al utilizar el modelo de observación usando esferas, no tiene sentido plantearse la utilización de la versión en 2D del algoritmo probabilístico, ya que como las celdas en las que calculamos la probabilidad dependen de los candidatos calculados por el algoritmo, las zonas de mayor probabilidad serán las mismas en ambos casos y el tiempo de cómputo del modelo será similar.

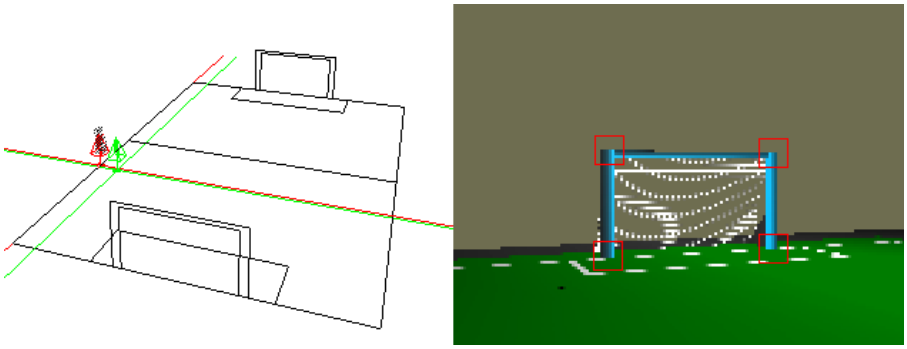
En este caso, hemos escrito un total de 540 líneas de código y el tiempo de ejecución necesitado por el algoritmo probabilístico al utilizar este modelo de observación ha sido de 67 milisegundos en la versión en 2D y de 160 en la 3D. A pesar de que los tiempos del modelo en 3D y 2D son prácticamente iguales, el tiempo total mostrado está influido por otros factores como la acumulación de la probabilidad, como veremos más adelante.

Para analizar el funcionamiento, vamos a mostrar las mismas imágenes que en el caso del modelo de observación con ángulos. Estas imágenes pueden verse en la figura 5.9.

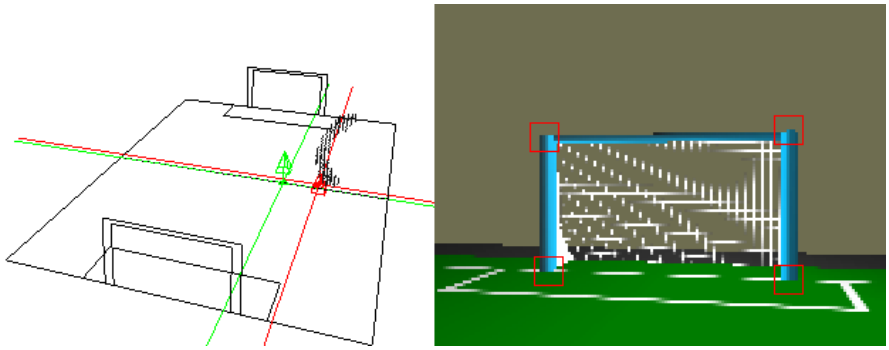
La precisión del modelo no es tan buena como en el de los ángulos, al igual que ya sucedía en la localización instantánea, siendo el error medio de 30.5 centímetros. Aunque tenemos que tener en cuenta de nuevo que se trata de una medición en 3D, mientras que en el modelo de observación anterior el error medio se tomó sobre la versión en 2D.



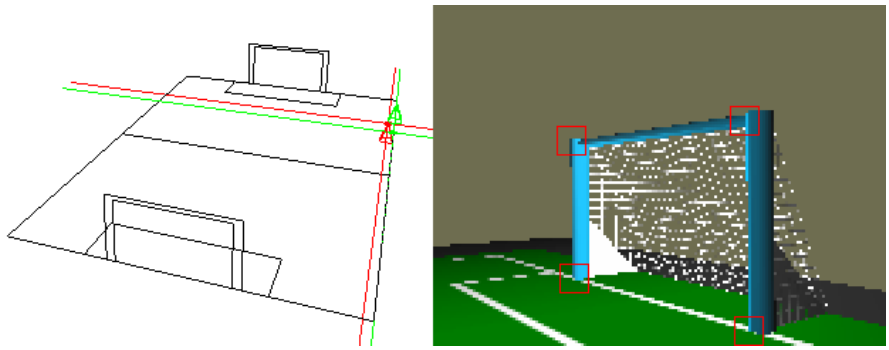
(a)



(b)



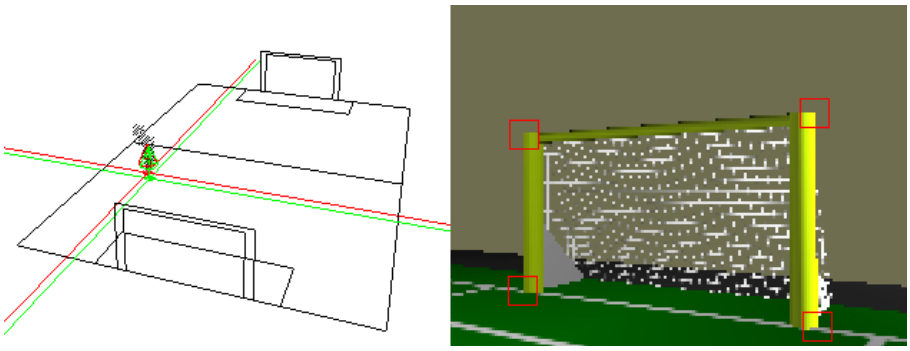
(c)



(d)

Figura 5.9: Localización instantánea usando esferas en Webots.

Como ya vimos, cuando vemos la portería amarilla los cálculos realizados son iguales que en la portería azul, con la diferencia de que al almacenar el candidato hay que cambiar sus coordenadas x e y para situarnos en la zona correcta del campo (figura 5.10).



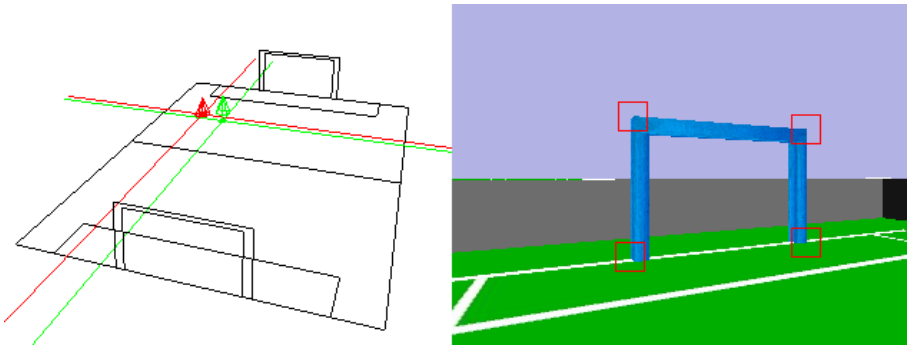
(a)

Figura 5.10: Localización probabilística usando esferas con portería amarilla.

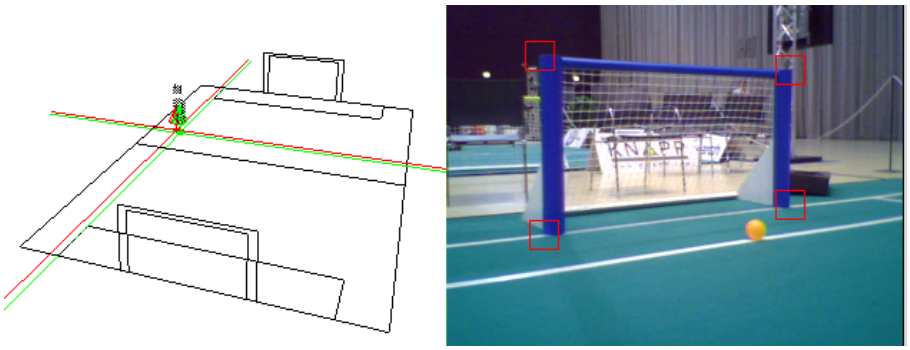
Vamos a mostrar ahora las imágenes obtenidas en el simulador Gazebo y con la cámara real del Nao, al igual que hicimos con el resto de algoritmos (figura 5.11).

Al ser este modelo de observación prácticamente igual al del algoritmo de localización instantánea, las ventajas e inconvenientes de ese algoritmo pueden aplicarse aquí, por ello, la precisión obtenida sigue siendo mejor en los laterales del campo que en la zona central, y disminuye la precisión cuando nos encontramos más lejos de la portería. En este caso la incertidumbre también aumenta con la distancia, aunque en menor medida que en el modelo anterior.

En el caso de que se produzcan pequeños errores en la detección, el cubo de probabilidad instantáneo calculado por el modelo puede no ser totalmente correcto, pero no cambiará bruscamente como sucedía al utilizar el algoritmo instantáneo. Además, al utilizar varias observaciones para calcular la probabilidad acumulada, podremos subsanar el error de una detección mal realizada en una única observación, haciendo que el algoritmo sea más robusto y fiable.



(a)



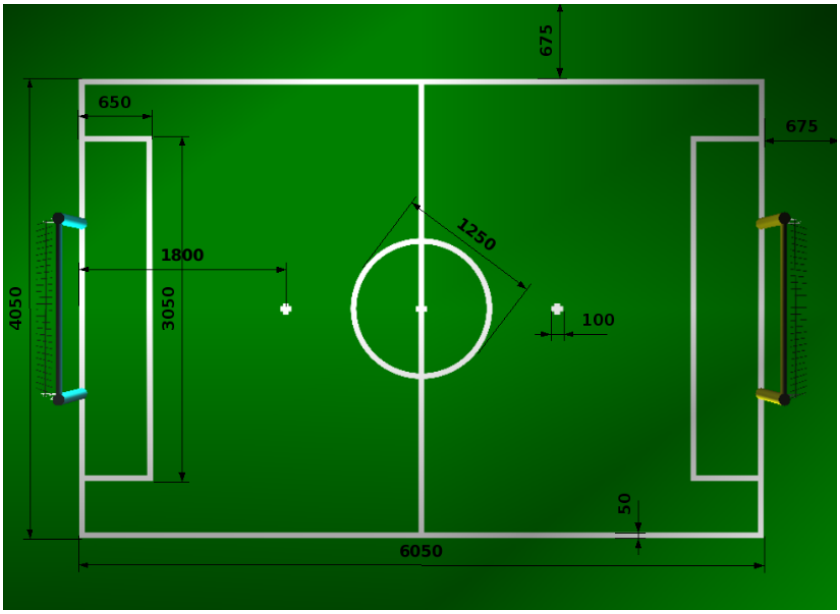
(b)

Figura 5.11: Imagen utilizando el simulador Gazebo (a) e imagen real tomada del Nao (b).

5.5. Localización probabilística basada en líneas

Las porterías, como ya hemos visto, son uno de los elementos más importantes a la hora de localizarnos dentro del campo, sin embargo, hasta ahora no hemos tenido en cuenta otro de los elementos que pueden ser de mucha ayuda, las líneas del campo. Con estas líneas se pueden utilizar técnicas similares a las descritas para la portería, y de esa forma aprovechar mejor la información recibida en las imágenes.

Las líneas del campo en el campo de la RoboCup están fijadas en el reglamento de la liga estándar, y cuyas dimensiones pueden observarse en la imagen 5.12.



(a)

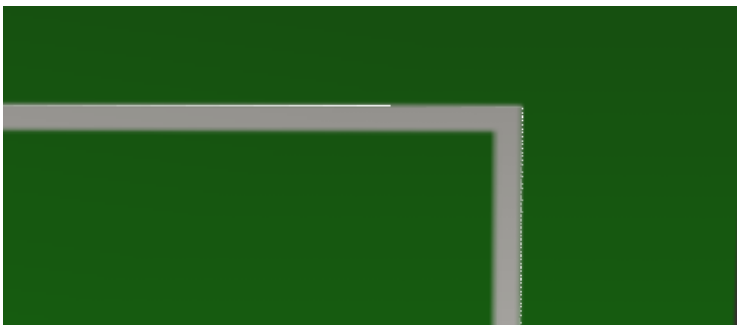
Figura 5.12: Líneas del campo establecidas en el reglamento de la RoboCup.

Si lográsemos detectar correctamente las líneas del campo, podríamos utilizarlo en los algoritmos anteriores. Por ejemplo, a la hora de realizar la detección probabilística utilizando ángulos, sólo necesitamos 2 puntos (los extremos de la portería) para poder calcular un ángulo α con el que determinar las distintas probabilidades.

Como puede verse, si encontrásemos dos puntos característicos de las líneas, podríamos utilizar el algoritmo anterior, porque sabemos las dimensiones de éstas.

Por ello, hemos decidido desarrollar un nuevo algoritmo encargado de detectar las intersecciones que se producen entre las líneas del campo, diferenciando además estas intersecciones según formen una L (figura 5.13(a)) o una T (figura 5.13(b)), para que nos den más información sobre la línea que estamos viendo.

(a)



(b)

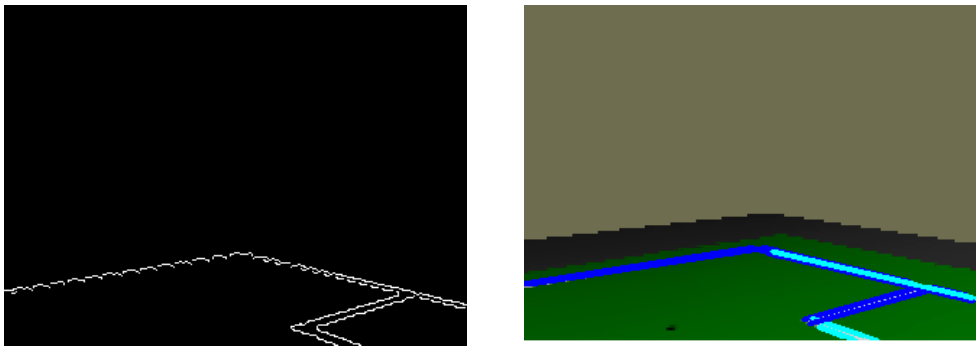
Figura 5.13: Intersección en forma de T (a) o en forma de L (b).

El algoritmo desarrollado realiza en primer lugar una serie de operaciones similares a la detección de la portería, primero realiza un filtro de color en HSV para filtrar todos los objetos de color blanco que se encuentren en la imagen. Una vez realizado esto se realiza un filtro Canny para detectar los bordes y posteriormente se utiliza la transformada de Hough para detectar las líneas en la imagen.

En la figura 5.14 puede verse un ejemplo de lo descrito, donde primero puede apreciarse la salida del filtro Canny (5.14(a)) y después el resultado obtenido en la transformada de Hough (5.14(b)).

El siguiente paso a realizar, es calcular las intersecciones que se producen entre todas las líneas obtenidas por la transformada de Hough. Este cálculo se describirá en detalle en la siguiente sección.

En principio estas intersecciones serían las que buscamos en el algoritmo, pero debido a problemas de percepción en la imagen o a otros factores, como que el ancho de la línea blanca es demasiado grande, puede que se detecten varias rectas dónde sólo hay una línea del campo.

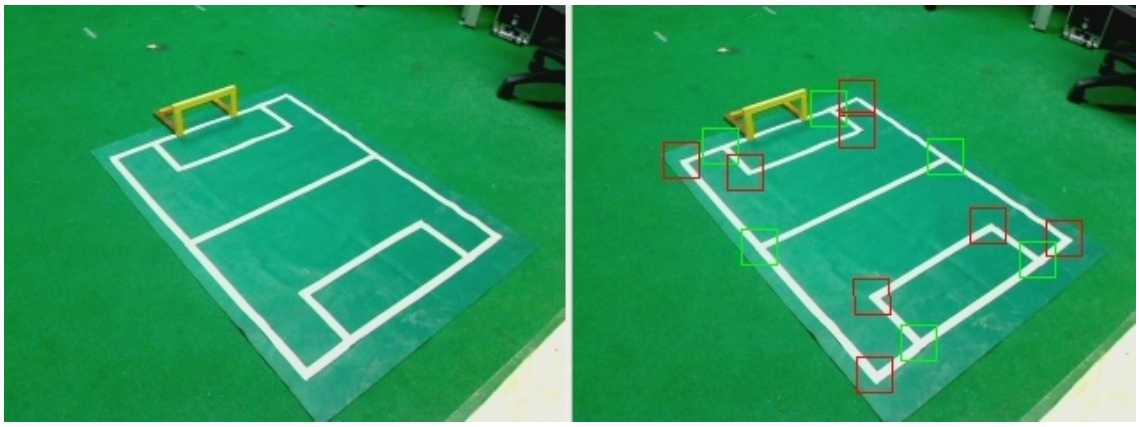


(a) (b)

Figura 5.14: Líneas tras filtro Canny (a) y transformada de Hough (b).

De modo que una vez obtenidas las intersecciones entre todas las líneas, realizamos un bucle buscando las intersecciones que se encuentren muy próximas (dentro de unos límites prefijados) y que se puedan unificar como una sola intersección. Además, al unificar varias intersecciones en una sola, se recalcula si se ha detectado una T o una L, como veremos más adelante.

Finalmente se representan en la imagen las intersecciones detectadas y se muestran en color rojo si se ha detectado como una L y de color verde si es una T, como puede verse en la figura 5.15.



(a)

Figura 5.15: Identificación de T's y L's en el campo.

5.5.1. Cálculo de intersecciones

Para calcular la intersección entre cada par de líneas detectadas, es necesario conocer la ecuación de la recta de cada línea, sin embargo, la información que nos da la transformada de Hough son los extremos de cada línea detectada en 2D. Para simplificar los cálculos a la hora de calcular las intersecciones hemos decidido utilizar la geometría proyectiva planar.

En la geometría proyectiva planar cada punto se representa en 3 coordenadas $[x, y, t]$, donde x e y se corresponden con las coordenadas del punto en 2D, y t es siempre 1 para homogeneizar.

Así la ecuación general de la recta en 2D que se muestra en 5.7, puede calcularse a partir de 2 puntos *P1* y *P2*, calculando los parámetros $[A, B, C]$ mediante el producto vectorial de ambos puntos (5.8). Con estas ecuaciones podemos conocer la ecuación general de la recta de cada línea, a partir de sus dos extremos.

$$Ax + By + C = 0 \tag{5.7}$$

$$P1xP2 = \begin{bmatrix} A & B & C \\ P1_x & P1_y & 1 \\ P2_x & P2_y & 1 \end{bmatrix} = \begin{bmatrix} P1_y - P2_y \\ P2_x - P1_x \\ P1_xP2_y - P1_yP2_x \end{bmatrix} \tag{5.8}$$

Una vez obtenidas estas rectas, se calculan todas las intersecciones entre cada par de líneas. El cálculo de estas intersecciones es muy simple utilizando geometría proyectiva planar, puesto que teniendo cada recta de la forma $[A, B, C]$, podemos calcular la intersección de dos rectas *R1* y *R2* en un punto de la forma $[x, y, t]$ mediante el producto vectorial, como en la ecuación 5.9.

$$R1xR2 = \begin{bmatrix} x & y & t \\ R1_A & R1_B & R1_C \\ R2_A & R2_B & R2_C \end{bmatrix} = \begin{bmatrix} R1_BR2_C - R1_CR2_B \\ R1_CR2_A - R1_AR2_C \\ R1_AR2_B - R1_BR2_A \end{bmatrix} \tag{5.9}$$

Una vez obtenido el punto P donde intersecan las dos rectas, debemos hacer una serie de comprobaciones para saber si realmente estas rectas intersecan.

Lo primero que hay que tener en cuenta para saber si la intersección en correcta, es ver si estas rectas son paralelas, esto puede saberse viendo el resultado obtenido en P_t, siendo las rectas paralelas si $P_t = 0$.

Además, puesto que las rectas identificadas en la imagen puede tener cierto error, comprobamos si las rectas son "casi" paralelas, esto se puede conseguir calculando la pendiente m y el ángulo α_m de ambas rectas con las ecuaciones 5.10.

$$\begin{aligned} m &= \frac{-R_A}{R_B} \\ \alpha_m &= tan(m) \end{aligned} \tag{5.10}$$

Podemos saber si las rectas son "casi" paralelas comparando los α_m de cada una de las dos rectas, para ello hemos calculado un valor α_d con la ecuación 5.11. Hemos considerado paralelas las rectas cuando α_d ha sido menor que un parámetro llamado *diffm*. Sobre este parámetro hemos realizado numerosas pruebas y finalmente hemos establecido un valor de 0.35, con el que los resultados han sido satisfactorios.

$$\alpha_d = |\alpha 1_m - \alpha 2_m| \tag{5.11}$$

Una vez comprobado que las líneas no eran paralelas (o casi paralelas) hemos considerado como válidas las coordenadas (x, y) del punto obtenido, pero hemos tenido que homogeneizarlas con respecto a t, como se ve en la ecuación 5.12.

$$\begin{bmatrix} x \\ y \\ t \end{bmatrix} = \begin{bmatrix} x/t \\ y/t \\ 1 \end{bmatrix} \tag{5.12}$$

El siguiente paso que realizamos, es calcular si la intersección que hemos calculado se sale de los límites de la imagen, para ello simplemente comprobamos si se sobrepasa el ancho y alto de las dimensiones de la imagen, lo que significaría que la intersección no es válida.

Lo siguiente que comprobamos es si la intersección calculada se encuentra dentro de los límites de las líneas, ya que al utilizar las ecuaciones de las rectas, éstas se propagan en el infinito y podemos obtener una intersección fuera de los límites. Estos límites son los que nos proporcionó la transformada de Hough, aunque para evitar errores de precisión, utilizamos un pequeño error de cálculo, dentro del cual lo consideramos válido.

Si las intersecciones se consideran válidas en todos los aspectos anteriores se guardan para posteriores comprobaciones.

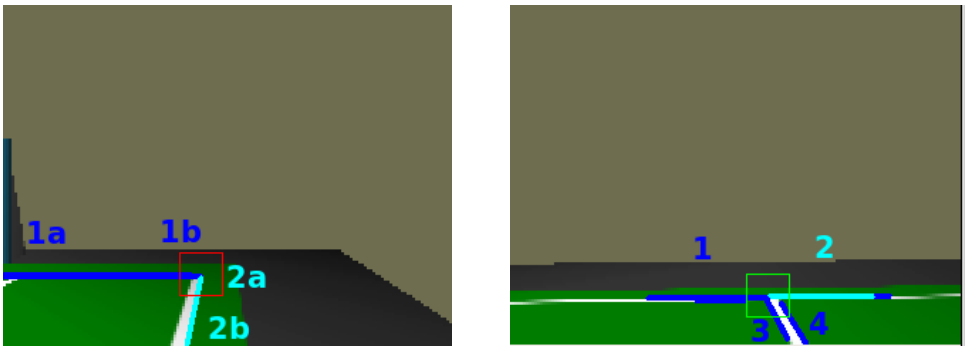
Detección de *T*'s y *L*'s

Una vez calculadas las intersecciones válidas, el siguiente paso es comprobar si lo que hemos detectado es una T o una L, para lo que realizamos dos comprobaciones.

Antes de explicar la primera comprobación, hay que tener en cuenta que las dos líneas dadas en cada intersección, están definidas por sus extremos de inicio y fin, así que consideramos que una intersección es una L siempre que el punto de intersección se produzca en uno de los dos extremos de cada línea, mientras que en cualquier otro caso será una T.

Así pues, la primera comprobación consiste en ver si la distancia desde el punto de intersección hasta alguno de los extremos de la primera línea es pequeña (dentro de unos rangos establecidos). Si ninguno de los extremos de la primera línea están junto a la intersección, se considerará que es una T, y en caso contrario, se realizará la comprobación sobre la segunda línea. Si en ambas líneas hemos encontrado que uno de los extremos está junto al punto de intersección, se considera una L.

En la imagen 5.16(a) podemos ver un ejemplo de lo anterior, donde tanto los extremos *1b* como *2a* están cerca del punto de intersección, por lo tanto es considerado como una L.



(a) (b)

Figura 5.16: Ejemplos de detección de una L (a) y una T (b).

Si con la comprobación anterior determinamos que estamos ante una T, el algoritmo no debe hacer más comprobaciones, sin embargo, si detectamos que es una L como en el ejemplo dado, debemos realizar una segunda comprobación.

En el caso de la imagen 5.16(b), con nuestra comprobación anterior habríamos obtenido que tanto las líneas 1 y 3, como las 2 y 4 forman por separado dos L, sin embargo podemos apreciar como en realidad se trata de una T. Para estos casos, una vez que ya hemos detectado todas las intersecciones que están en una misma zona, y que consideramos como se tratan de la misma intersección, realizamos una nueva comprobación.

Recorremos cada par de rectas cuyo punto de intersección sea el mismo, como ya vimos en la sección anterior, y sobre este punto de intersección creamos un cuadrado imaginario similar al dibujado en 5.16(b), donde comprobamos si las líneas que han intersecado en ese punto sobrepasan los límites superior, inferior, derecho e izquierdo.

Tras recorrer todas las líneas, miramos qué límites han sido sobrepasados, y en el caso de que se rebasen los límites superior e inferior, o izquierdo y derecho, se habrá encontrado una T.

En la imagen 5.16(b), a pesar de que 1-3 y 2-4 forman L's por separado, al realizar esta segunda comprobación, y puesto que intersecan en el mismo punto, los límites derecho, inferior e izquierdo del cuadrado imaginario se sobrepasan, con lo que encontramos una T.

5.5.2. Modelo de observación con líneas y ángulos

El modelo de observación visto en la sección 5.3 puede ser también aplicado a las líneas del campo de forma similar, puesto que sólo necesita de dos puntos conocidos para calcular un α teórico, que se corresponderán con los extremos de cada línea.

En este caso, el modelo de observación recibirá como entrada todas las intersecciones detectadas en la imagen, para las cuales se conocerá tanto su posición en la imagen como el tipo de intersección (si se trata de una T o una L).

En el caso de la portería, conocemos cuál de las dos porterías estamos viendo, y por lo tanto conocemos cuál es su posición dentro del campo en coordenadas absolutas, algo que no ocurre con las líneas, puesto que nos es imposible distinguir cuál de las líneas del campo estamos viendo.

Gracias a la detección de T's y L's, podemos diferenciar algunas de las líneas, pero aún así podemos encontrarnos 3 tipos de líneas distintas, que son:

- Líneas donde ambos extremos son una T (como la línea que separa los campos)
- Líneas donde ambos extremos son una L (como sucede en las líneas del área grande)
- Líneas con un extremo en forma de L y otro en forma de T(como por ejemplo en las líneas laterales del campo).

Además, al encontrar en la imagen una serie de intersecciones, no podemos saber qué intersecciones son las que corresponden a cada línea, de modo que si tenemos tres

puntos *A*, *B* y *C*, correspondientes a las rectas *A*-*B* y *B*-*C*, también debemos comprobar la recta *A*-*C*, a pesar de no existir en la realidad.

Así, para calcular la probabilidad en cada celda del cubo de probabilidad instantáneo, debemos seguir una serie de pasos:

La probabilidad de la celda se obtendrá a partir de la probabilidad de cada par de puntos. A su vez, esta probabilidad de cada par de puntos se calculará a partir de la probabilidad de cada posible línea que corresponda con ese par de puntos.

Esto es debido, a que para poder utilizar la ecuación 5.4, necesitamos calcular un α real y un α teórico. El α real puede obtenerse a partir del par de puntos que estemos utilizando, pero para el α teórico necesitamos conocer la posición absoluta de la recta que estamos viendo, que como ya hemos dicho no es posible.

Por ello, debemos calcular un α teórico para cada una de las posibles líneas que estemos viendo. Esto lo hacemos comparando el tipo de línea que tenemos con todas las posibles líneas que existen en el campo.

Para cada α teórico calculamos la diferencia respecto al α real igual que vimos en la ecuación 5.4, y calculamos su probabilidad con la ecuación que en 5.5.

Si la probabilidad calculada es mayor que 0.5, anotaremos un punto positivo a este par de puntos, y en caso contrario apuntaremos uno negativo. De modo que, al realizar lo mismo para cada α teórico tendremos P puntos positivos y N puntos negativos, así pues la probabilidad de ese par de puntos la calcularemos con la ecuación 5.13, donde *Total* es el número de αs teóricos que hemos evaluado.

$$p = 0{,}5 + P * 0{,}1 - \frac{N * 0{,}1}{Total - 1} \tag{5.13}$$

Con esto ya habremos obtenido la probabilidad para un par de puntos.

Así, calcularemos la probabilidad de la celda, realizando la misma operación para cada par de puntos que encontremos, y acumulando estas probabilidades parciales mediante la ecuación 5.1.

5.5.3. Experimentos

Al igual que sucedía con el modelo de observación con porterías y ángulos, vamos a utilizar la versión en 2D del algoritmo probabilístico, para simplificar la solución y mejorar el rendimiento obtenido.

Este modelo de observación nos ha ocupado unas 160 líneas de código, con un tiempo de ejecución del algoritmo de probabilidad de 52 milisegundos cuando se detectan 2 intersecciones, 80 milisegundos cuando se detectan 3 y aumentando progresivamente el tiempo según el número de intersecciones encontradas. Dentro de este tiempo de ejecución, el tiempo que se tardan en detectar las distintas intersecciones es de unos 19 milisegundos independientemente del número de intersecciones detectadas (dentro de unos valores razonables).

Vamos a mostrar el funcionamiento de este modelo con una imagen tomada en el simulador Webots (5.17(a)). Como podemos ver, hay 4 zonas simétricas donde la probabilidad es muy similar, y en una de ellas se encuentra realmente nuestro robot.

La selección final (flecha roja) realizada por el algoritmo es sólo anecdótica, puesto que no es posible determinar la posición a partir de la observación tomada, ya que puede corresponder a cualquiera de las 4 zonas del campo indistintamente.

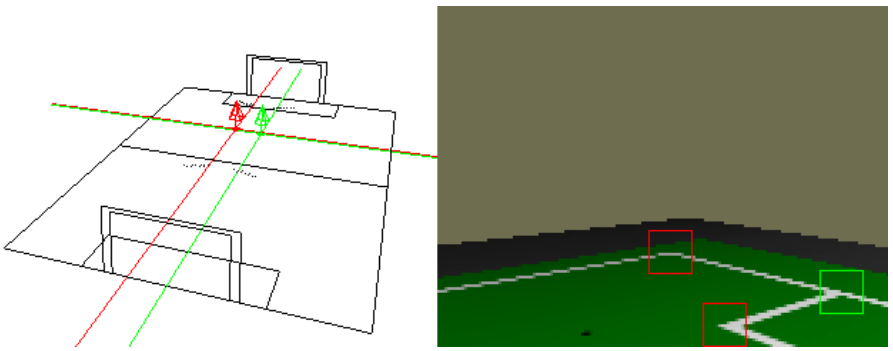
En el siguiente ejemplo (5.17(b)), podemos ver como al tener tan poca información, tan solo 2 puntos, es difícil conocer la localización en la que nos encontramos, ya que hay muchas zonas en el campo en las que podemos obtener esa recta. Por ello, en este caso la localización dada es poco fiable, y necesitaríamos otras observaciones para determinar la zona donde nos encontramos.

Como se puede comprobar, las líneas del campo no nos permiten conocer nuestra posición exacta por si solas, puesto que siempre existirán simetrías a lo largo del campo, pero nos indicarán ciertas zonas en las que podemos encontrarnos, de modo que cuando veamos otras líneas o la portería, podremos acabar sabiendo dónde nos encontramos gracias a la acumulación de observaciones.

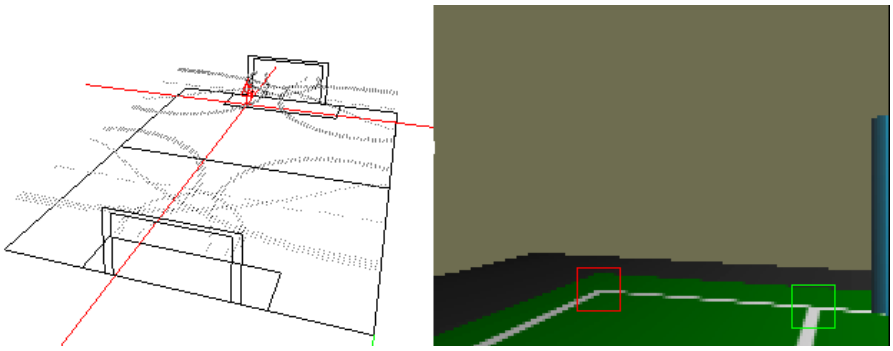
Según hemos observado, al aumentar el número de intersecciones detectadas las zonas del campo con probabilidad alta se reducen, lo que puede comprobarse estudiando las dos imágenes anteriores, donde las zonas con alta probabilidad han sido mucho mayores cuando sólo hemos detectado dos líneas que cuando hemos detectado tres.

Si se produce un error en la detección de las líneas, el error cometido en la localización puede ser muy alto, puesto que si nos equivocamos al detectar una intersección de tipo T y lo catalogamos como tipo L o viceversa, los resultados del modelo podrán cambiar totalmente, siendo la localización resultante errónea.

Aunque hay que tener en cuenta, que al igual que en el resto de modelos, al tratarse de un algoritmo que necesita varias observaciones para determinar la localización, podemos soportar varias observaciones incorrectas antes de cometer un error en la localización.



(a)



(b)

Figura 5.17: Probabilidad utilizando las líneas del campo.

5.6. Experimentos

Hasta ahora hemos comprobado experimentalmente el funcionamiento de cada uno de los modelos de observación por separado, sin embargo, los resultados de estos modelos estas influenciados por la forma en que acumula las observaciones el algoritmo probabilístico.

En este sentido, las líneas de código que ha ocupado la parte que se encarga de la acumulación es de 190 líneas, y los tiempos de ejecución en los que se produce la fusión son de 5 milisegundos en la versión 2D y de 100 milisegundos en la 3D, lo que explica algunos de los resultados obtenidos en los modelos anteriores.

Para poder comparar mejor el funcionamiento de los algoritmos de probabilidad, en esta sección vamos a realizar primero una comparativa entre los modelos de observación ya vistos, después vamos a analizar la acumulación de observaciones y por último, compararemos los algoritmos de probabilidad con los instantáneos que vimos en el capítulo anterior.

5.7. Comparativa entre los modelos de observación

Al igual que hicimos en el capítulo 4, vamos a comparar los modelos de observación vistos en este capítulo para utilizar el más adecuado.

La complejidad en la realización de los modelos ha sido mayor en el modelo de observación usando esferas que en el de los ángulos, ocupando el desarrollo del segundo mucho más que el primero. Al igual que sucedía en la localización instantánea, el modelo con ángulos utiliza una solución analítica, haciendo que la probabilidad en cada celda sea fácil de definir, mientras que el modelo de las esferas hace uso de la función de coste para calcular la probabilidad, dependiendo la fiabilidad del modelo de esta función de coste.

En cuanto al tiempo de ejecución, si tenemos en cuenta la version 2D del modelo de observación con ángulos con el modelo en 3D con esferas, el rendimiento es menor en el caso del primero (54 milisegundos frente a 160). Si bien este resultado está influido por la acumulación de observaciones como hemos visto, y en caso de utilizar la versión en 3D del modelo de observación con ángulos, el tiempo de ejecución es mucho mayor que para las esferas.

La precisión del primero de los modelos es mayor, aunque como hemos dicho hay que tener en cuenta que uno es en 2D y el otro en 3D. Al contrario de lo que sucedía en el del algoritmo instantáneo, esta vez podemos utilizar una versión en tres dimensiones también

en el modelo de los ángulos, por lo que eso no se puede contar como un factor a favor del modelo de las esferas. Aunque si utilizamos la versión en 3 dimensiones, la precisión del modelo de los ángulos empeorará y su tiempo de ejecución será mucho mayor.

Otro aspecto a tener en cuenta, es que hemos podido crear un nuevo modelo de observación utilizando las líneas del campo a partir del modelo con porterías y ángulos, aunque como hemos visto, es difícil localizar al robot utilizando únicamente este modelo debido a las simetrías en el campo.

5.7.1. Acumulación de observaciones

Durante el desarrollo del proyecto, hemos necesitado conocer el funcionamiento de nuestros algoritmos de probabilidad, para saber si tanto las probabilidades instantáneas como las acumuladas se estaban realizando correctamente.

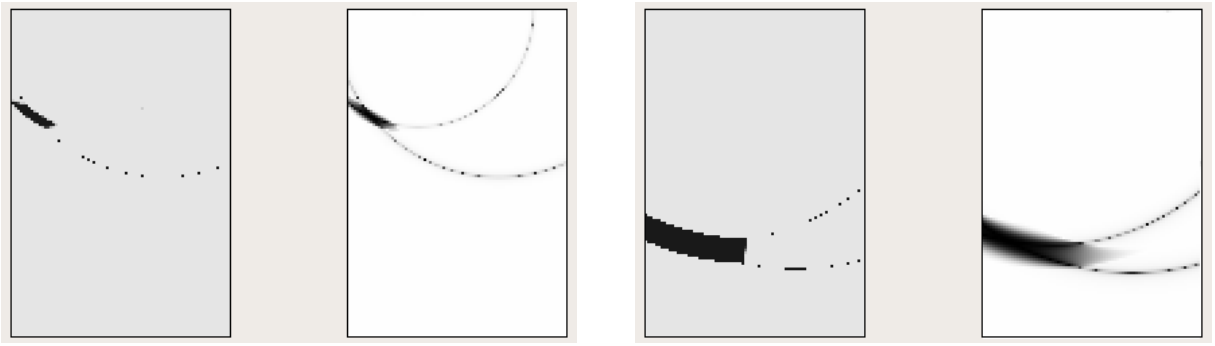
Para eso, creamos una funcionalidad que nos permitiese conocer en tiempo real cuál era la probabilidad que se estaba utilizando, guardando en una imagen con formato PPM los cubos de probabilidad en 2D generados.

Durante la ejecución de los algoritmos, podíamos llamar a esta funcionalidad y obteníamos tanto la probabilidad instantánea como la acumulada, lo que nos permitía comprobar la acumulación de las observaciones a lo largo del tiempo. En la imagen 5.18 se muestran dos ejemplos, donde la figura 5.18(a) se corresponde con la imagen que vimos en 5.3(a), y la figura 5.18(b) corresponde con la imagen 5.3(b).

En ambas figuras, pueden apreciarse dos imágenes distintas, la de la derecha se corresponde con la probabilidad instantánea de la última observación, mientras que la de la izquierda es la probabilidad acumulada a lo largo del tiempo.

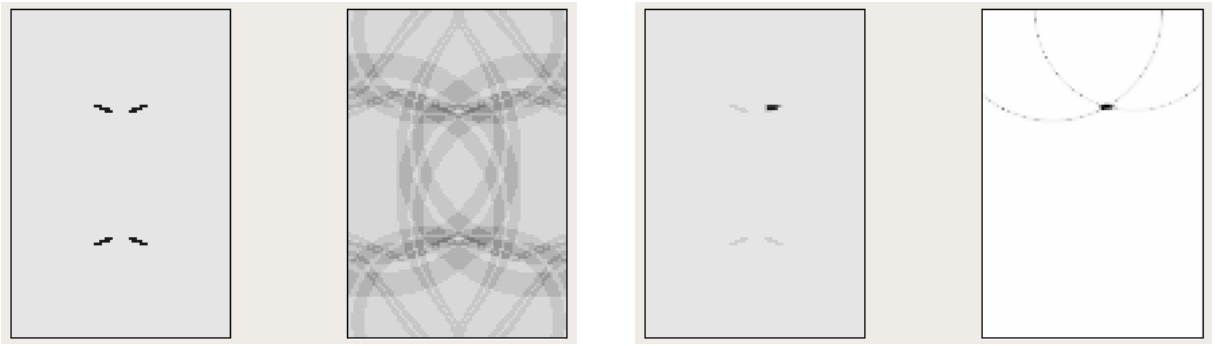
Esta misma funcionalidad, también se ha podido utilizar cuando lo que observábamos eran las líneas, sirviendo para ver de una manera más clara las distintas simetrías que se producían a lo largo del campo. Así, en la figura 5.19(a) puede verse la captura en 2D correspondiente a la imagen que vimos en 5.17(a), y donde pueden verse con mayor claridad las zonas en las que la probabilidad acumulada eran mayores.

Otra utilidad de esta funcionalidad, es ver la evolución de la probabilidad. Así, a partir de la probabilidad acumulada que hemos visto en 5.19(a), si en ese momento pasábamos de ver las líneas a volver a mirar a la portería desde la misma posición, las probabilidades obtenidas eran las que se muestran en 5.19(b).



(a) (b)

Figura 5.18: Imágenes PPM con la probabilidad.

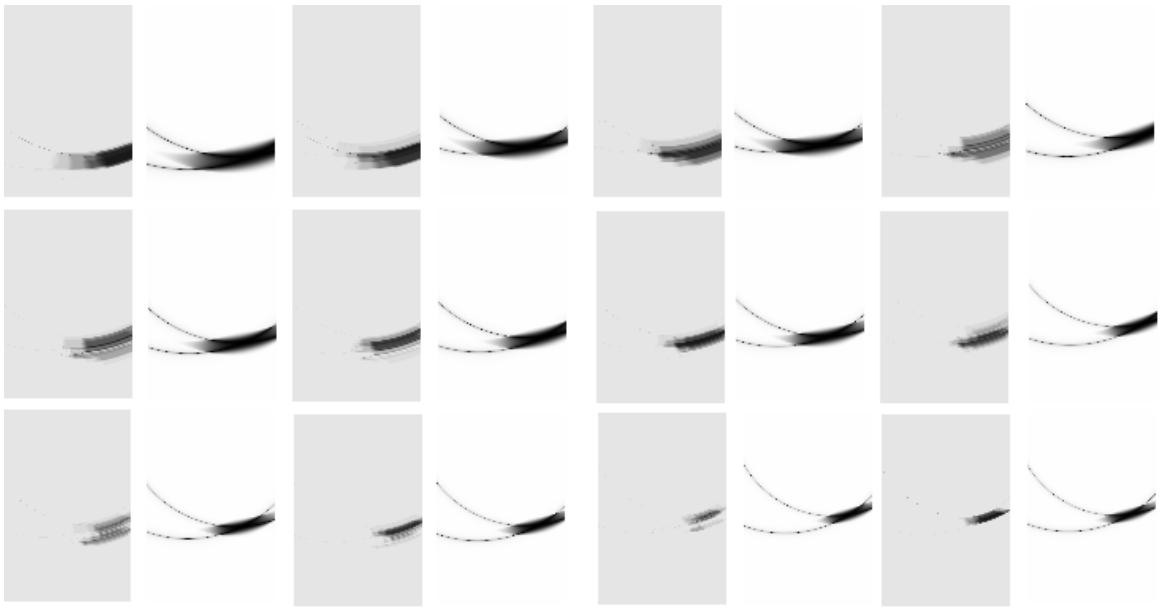


(a) (b)

Figura 5.19: Imagen PPM al usar las líneas (a). Acumulación al ver la portería (b)

Puede verse en esta imagen, como a partir de la probabilidad instantánea, cambia la probabilidad acumulada, manteniéndose con mayor probabilidad la zona en la que se encuentra realmente el robot y bajando la probabilidad en el resto de sitios.

En la imagen 5.20 mostramos la evolución en la acumulación que se ha producido al movernos con el robot simulado en Gazebo, teniendo en cuenta que entre cada captura mostrada han pasado varias iteraciones y que se ha utilizado el modelo de observación con porterías y ángulos en 2D. En esta imagen se van alternando las probabilidades acumuladas (izquierda) con las instantáneas (derecha), y puede verse como va cambiando poco a poco la probabilidad acumulada mientras avanza el robot, y cuando éste se para, la probabilidad acaba convergiendo.



(a)

Figura 5.20: Evolución en la acumulación.

5.8. Comparativa entre algoritmos instantáneos y probabilísticos

Comparando los dos métodos de localización, la localización instantánea nos proporciona rapidez al determinar la posición del robot, al contrario que la localización probabilística, y puede ser válida para una primera aproximación de nuestra situación actual.

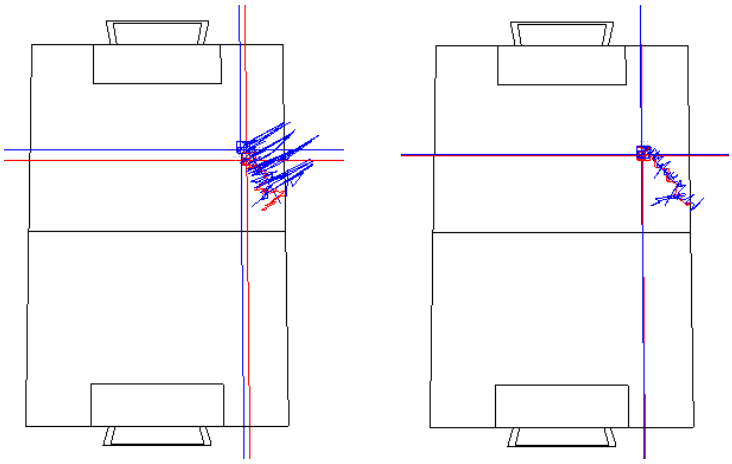
Sin embargo, si queremos una localización más fiable, los algoritmos probabilísticos se comportan mucho mejor que los instantáneos, y además permiten seguir dando una posición fiable aunque la detección de los elementos en una imagen momentánea sea errónea, algo que no sucede con los algoritmos instantáneos, ya que sólo se basan en la última imagen obtenida. Esto podría suceder cuando la detección de las porterías no se realiza correctamente o cuando se producen oclusiones.

Como contrapartida, además de la eficiencia que ya hemos comentado, la localización probabilística nos da la posición del robot con cierto retardo, al necesitar varias imágenes hasta llegar a una conclusión.

Para poder comparar mejor los distintos algoritmos, hemos añadido a las localizaciones mostradas en OpenGL una estela que nos muestra el camino seguido por el robot a lo largo del tiempo. Además hemos permitido la ejecución simultanea de cada modelo de observación probabilísticos con su algoritmo instantáneo semejante.

En la figura 5.21 mostramos la comparación entre los dos tipos de algoritmos desarrollados. En la imagen de la izquierda podemos ver la comparación entre el algoritmo instantáneo usando toros y el algoritmo probabilístico usando el modelo de observación con porterías y ángulos, mientras que en el de la derecha vemos la comparativa entre los dos tipos de algoritmos utilizando esferas.

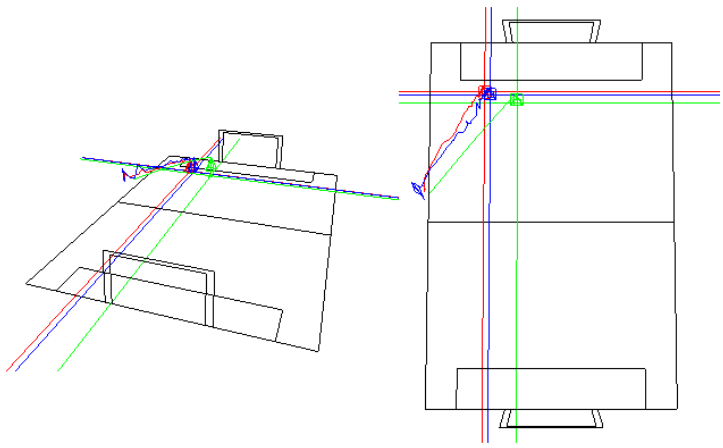
Las dos imágenes han sido tomadas siguiendo trayectorias similares en el simulador Webots. Podemos ver como los algoritmos probabilísticos (rojo) son más estables que los instantáneos, ya que siguen una trayectoria más suave.



(a) (b)

Figura 5.21: Comparación de algoritmos en Webots.

Para comparar estos dos algoritmos con la posición real, podemos ver la figura 5.22, donde se pueden ver las trayectorias seguidas por ambos algoritmos respecto a la posición real.



(a)

Figura 5.22: Comparación de algoritmos en Gazebo.

Capítulo 6

Conclusiones y Trabajos futuros

En los capítulos anteriores hemos explicado el algoritmo diseñado para resolver el problema de la localización de un robot en del campo de la RoboCup. Con este capítulo vamos a resumir las conclusiones más importantes que hemos sacado con la realización de este proyecto y repasaremos los objetivos planteados para saber en que grado se han satisfecho. Por último, indicaremos cuáles pueden ser los trabajos futuros abiertos en relación con el proyecto realizado.

6.1. Conclusiones

Tras un desarrollo de cerca de 5 mil líneas de código, se han alcanzado los objetivos planteados en el capítulo 2, cuyo objetivo principal era el de conseguir localizar al robot Nao en el campo de la RoboCup utilizando exclusivamente la percepción visual de las porterías.

A continuación vamos a repasar los distintos subobjetivos que se plantearon para conocer la solución adoptada en cada uno de ellos:

1. El primero de los subobjetivos a alcanzar, era la detección de las porterías en la imagen (descrita en el capítulo 4.2), que se ha resuelto utilizando la librería OpenCV, la cual nos ha proporcionado diversas funciones muy útiles para la detección, como el filtro Canny o la transformada de Hough. Como ya vimos, se resolvió la detección de la portería de dos modos distintos, quedándonos con el método más robusto, y que destacaba por ser independiente a los giros de la cámara.

Además, para facilitar la detección de las porterías, hemos calculado un horizonte a lo largo del campo que nos sirve para evitar que confundamos los colores de la portería con los de otros elementos que se encuentren en el campo.

2. El siguiente subobjetivo era lograr localizar nuestro robot en 3D a partir de una única observación obtenida por la cámara. Esto se ha cumplido con el desarrollo de los algoritmos de localización 3D instantánea descritos en el capítulo 4. Como ya se vio, se hizo un uso intensivo de la geometría 3D para calcular la posición del robot a partir de figuras geométricas como los toros y las esferas.

3. Los algoritmos anteriores se adaptaron para poder utilizar varias observaciones para calcular la posición del robot, utilizando para ello probabilidad, cumpliéndose así otro de los subobjetivos planteado. En este caso, se desarrollaron varios modelos de observación probabilísticos utilizando las porterías, que calculaban la probabilidad en todo el campo para que después se acumulase con la probabilidad ya calculada mediante fusión Bayesiana.

   Además, a pesar de no ser un objetivo del proyecto, se realizó otro modelo de observación que utilizaba para el cálculo de la probabilidad las líneas del campo. Para lo que tuvimos que desarrollar un algoritmo encargado de detectar las intersecciones que formaban las líneas que se veían en la imagen, dividiéndolas en T's y L's, según el tipo de intersección.

4. Los algoritmos desarrollados han sido validados experimentalmente tanto en los simuladores (Webots y Gazebo), como se especificaba en los objetivos, como con imágenes reales obtenidas del robot Nao. Con la utilización tanto de los simuladores como de las imágenes reales, hemos podido validar las técnicas utilizadas en entornos distintos, mejorando la robustez del sistema.

   Estos experimentos han demostrado que es viable aplicar las técnicas desarrolladas en el contexto real de la RoboCup, sirviendo de alternativa a las técnicas que se aplicaban hasta ahora.

5. El último de los subobjetivos era la contar con una interfaz gráfica de usuario, que se ha creado mediante GTK+ y Glade y que permite manejar del sistema de forma sencilla e intuitiva. Asimismo, esta interfaz incorpora una escena en 3D creada en OpenGL donde se muestra la localización calculada en cada momento.

Requisitos

Los requisitos que han tenido que cumplir nuestras aplicaciones han estado marcados por lo especificado en el capítulo 2.2.

Como ya establecimos en estos requisitos, se ha utilizado la arquitectura de desarrollo JdeRobot para generar nuestras aplicaciones, siendo finalmente los lenguajes de programación C y C++ los utilizados. El hecho de utilizar esta plataforma de desarrollo nos ha facilitado enormemente el desarrollo, al disponer de librerías y drivers ya desarrollados que han sido de gran utilidad. Éste es el caso de la librería Progeo, utilizada en los cálculos geométricos, o el driver *Gazebo*, que nos ha servido para comunicarnos con el simulador de su mismo nombre.

Las aplicaciones realizadas para JdeRobot, han sido el esquema *Location*, donde podemos seleccionar cada uno de los algoritmos descritos de una forma sencilla, y el driver *Naobody*, para la comunicación con el robot Nao simulado utilizando la arquitectura software Naoqi.

También han sido mérito de este proyecto, la realización de los esquemas *Headtracking* y *Opencvdemo*, desarrollados durante la fase de aprendizaje de la plataforma y de visión artificial, como ya se describió en el capítulo 3.

El desarrollo de estos esquemas y drivers nos han permitido manejar múltiples bibliotecas, que son de gran utilidad en campos muy diferentes, como OpenGL para la representación en 3D, GSL, utilizada en los cálculos matriciales, GTK, para el desarrollo de interfaces gráficas, etc.

Además, la plataforma JdeRobot se ha enriquecido gracias a nuestro proyecto, puesto que el esquema *Opencvdemo* y el driver *Naobody* ya forman parte de la versión oficial de la distribución.

En cuanto a los algoritmos realizados, se ha utilizado únicamente una cámara como sensor para conocer nuestra localización, recibiendo de este sensor imágenes de distintos tamaños. Al utilizar solamente una cámara, nuestros algoritmos serán aplicables a otros robots que cuenten con este sensor, independientemente de sus características.

La robustez de los algoritmos es muy alta siempre que no existan oclusiones y se vea a la portería de forma completa, siendo el algoritmo independiente de la inclinación de la cámara. En cuanto a la precisión, todos los algoritmos cuentan con un valor medio de error inferior al que establecimos en los requisitos, siendo el error medio de los algoritmos basados en toros de unos 10 cm, y el de los basados en esferas de unos 30 cm.

Por último, el requisito de la eficiencia de los algoritmos ha sido tenido en cuenta en todo momento mientras desarrollábamos el proyecto, por lo que siempre se han intentado desarrollar algoritmos cuya eficiencia fuese la mayor posible. Debido a lo anterior, nuestros algoritmos están listos para llevarse al robot Nao real, aún a pesar de sus limitaciones hardware, habiendo avanzado en este sentido con la creación del driver *Naobody*, totalmente compatible con el robot Nao real.

Novedades respecto a otros proyectos

Como ya se comentó en el capítulo 1, en el grupo de robótica de la URJC se han desarrollado a lo largo de los años numerosos proyectos relacionados con la localización y con la RoboCup.

Sin embargo, nuestro proyecto ha sido el primero en utilizar los robots actualmente utilizados en la plataforma estándar de la RoboCup, los robots humanoides Nao, a diferencia de otros proyectos como [Crespo, 2003] ó [Peño, 2003], que utilizaban robots EyeBot. Esto ha hecho que hayamos tenido dificultades en la comunicación con el robot, puesto que *Naoqi* aún se encuentra en sus fases iniciales, lo que ha sido subsanado con el desarrollo del driver *Naobody*.

Otro factor importante y que nos distingue de otros proyectos, como por ejemplo del proyecto [Kachach, 2005], es que para la localización sólo se ha utilizado una única cámara, sin poder hacer uso de otros sensores que nos hubiesen proporcionado otro tipo de información.

Otro de los puntos a tener en cuenta, ha sido que la localización se ha realizado en 3D, mientras que hasta ahora sólo se había abordado la localización en 2D, en proyectos como el de [López Fernández, 2005], con el aumento de complejidad que esto conlleva.

6.2. Trabajos futuros

El trabajo realizado en la realización del proyecto puede ser continuado más adelante por otras vías que describiremos a continuación.

Robot Nao real

La principal continuación que se puede realizar a partir de este proyecto, es la implementación de nuestros algoritmos en el robot Nao real. Teniendo en cuenta que casi todo nuestro desarrollo se ha realizado utilizando simuladores, y a pesar de que hemos tratado de hacer los algoritmos lo más eficiente posibles, es posible que al utilizar el robot Nao real, donde la capacidad de cálculo es muy limitada como ya se mostró en el capítulo 3.1, sea necesario aumentar la eficiencia de los algoritmos, ya sea optimizándolos o haciendo que el número de comprobaciones realizadas sea menor, perdiendo de ese modo precisión pero aumentando la eficiencia.

Este aspecto habrá que tenerlo muy en cuenta sobre todo en los algoritmos de probabilidad, ya que en la localización instantánea el tiempo de cálculo necesario es razonablemente bajo. Por ejemplo, en el cubo de probabilidad desarrollado para los algoritmos probabilísticos, pueden reducirse el número de celdas, haciendo que cada celda abarque un conjunto mayor de posiciones reales.

Filtros de color

Otro aspecto a tener en cuenta y que se podría mejorar, son los filtros de HSV utilizados, ya que al utilizar terrenos de juego simulados la luminosidad se mantenía constante, y nos ha bastado con fijar unos rangos para cada color que funcionen en todas las situaciones. Sin embargo al utilizar el robot real, donde la luminosidad puede variar según el lugar donde se realice la prueba, o incluso según la posición del robot dentro del campo de la RoboCup, habrá que plantearse la realización de los filtros HSV de un modo adaptativo, para que se adecúe a las condiciones del entorno. O incluso, puede estudiarse el trabajar con el modelo de color YUV, que es el formato nativo de los Nao, lo que nos ahorraría tiempo en la conversión de formatos.

Porterías incompletas

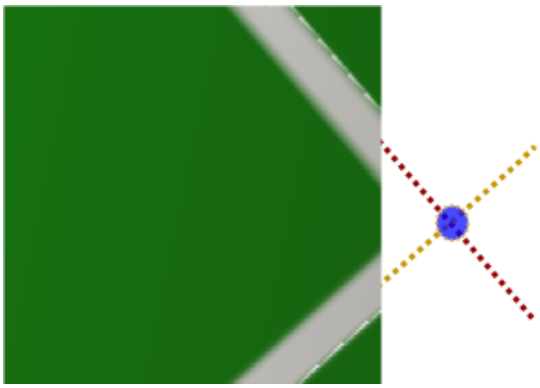
En los algoritmos que hemos desarrollado siempre se ha puesto como condición que las porterías del campo se viesen de forma completa. Por lo tanto, se podría explorar la vía de

la localización cuando sólo viésemos una parte de la portería. Esto sería posible, puesto que si sólo viésemos uno de los postes, no tendríamos tanta información como con la portería completa, pero ya se establecerían restricciones geométricas respecto a la localización de la cámara dentro del campo.

Líneas del campo

Otra vía futura que puede desarrollarse es la utilización de las líneas en los distintos algoritmos. Hasta el momento, hemos utilizado esta característica extra en la localización probabilística basada en toros, sin embargo es posible que pueda ampliarse al resto de algoritmos desarrollados haciendo leves modificaciones.

Relacionado con lo anterior, una técnica no utilizada y que puede abordarse, es la suposición de nuevas intersecciones entre líneas a partir de segmentos no completos en la imagen. Esto se puede ver mejor en la figura 6.1, donde contamos con 2 segmentos cuya visualización no es completa, pero la imagen nos da la suficiente información para suponer dónde se realizaría la intersección. Este punto de vista nos permitiría obtener nuevas intersecciones que no se pueden encontrar con nuestro algoritmo de detección de líneas actual, y nos permitiría mejorar la localización probabilística.



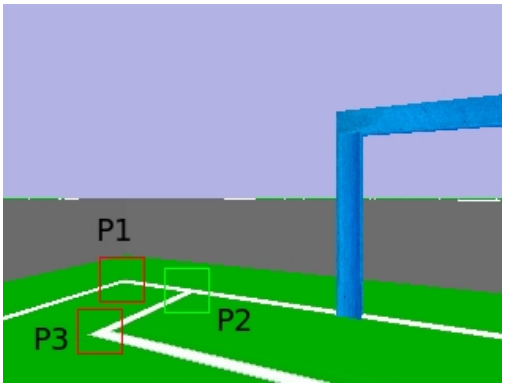
(a)

Figura 6.1: Búsqueda de intersecciones con vistas de segmentos parciales.

Por último, una nueva mejora que se puede realizar a nuestros algoritmos, es modificar el punto de vista de la detección de líneas. En nuestros algoritmos actuales detectamos puntos de intersección y los identificamos como T's y L's, de modo que al comparar estas intersecciones con las existentes en la realidad nos permiten conocer qué líneas podemos estar viendo. Sin embargo, si tenemos una situación como la de la imagen 6.2, al detectar en la imagen los puntos *P1*, *P2* y *P3*, no aprovechamos que *P1*-*P2* forman un segmento,

mientras que *P2-P3* forman otro, por lo que en nuestro algoritmo también tenemos que probar con el segmento *P1-P3*, aunque en la realidad este segmento no exista.

Para solucionar lo anterior, se podría modificar el algoritmo de detección de líneas, para que en lugar de guardar las intersecciones obtenidas, obtenga directamente segmentos completos, lo que nos permitiría evitar casos como el anterior.



(a)

Figura 6.2: Ejemplo de detección de intersecciones utilizando el simulador Gazebo.

Apéndice A

Headtracking

A modo de introducción a la arquitectura JdeRobot, desarrollamos un esquema que nos permitiese tomar un primer contacto con la plataforma y nos diese algunas bases de cara al proyecto final.

Esta primera aplicación estaba basada en el proyecto [Chung Lee, 2008], y consistía en poder movernos alrededor de una escena en 3D utilizando para ello el movimiento de nuestro propio cuerpo.

Para conseguir esto, utilizamos un Wiimote[1], un dispositivo capaz de detectar luces infrarrojas y de indicarnos la posición en la que se encuentran desde su punto de vista.

Para localizar nuestra posición, era necesario disponer de dos emisores de luz infrarroja colocados a una distancia previamente conocida. Una vez que el Wiimote nos daba la información sobre la posición de cada luz, y conociendo la distancia entre ellos, podíamos saber nuestra posición en (x, y, z) mediante cálculos trigonométricos.

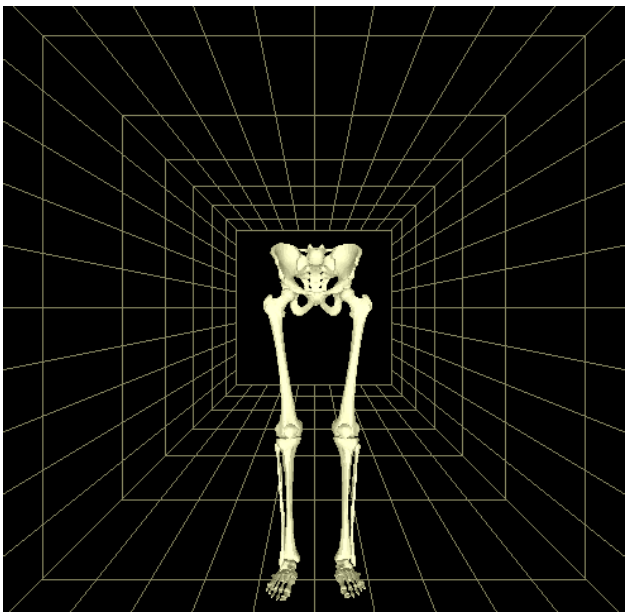
Para facilitar la colocación de las luces infrarrojas, creamos unas gafas con una luz infrarroja en cada extremo, de modo que conociésemos la distancia entre las luces, como puede verse en la imagen A.1.

La escena en 3D la creamos utilizando OpenGL y un esqueleto humano creado en el proyecto [Muelas Sahagún, 2009]. Sobre esta escena (figura A.2), usábamos las funciones de OpenGL para situar nuestro punto de vista en función de la posición calculada anteriormente, de modo que la impresión final era la de movernos alrededor de la escena. Obteniendo como resultado una interfaz gráfica inmersiva que ofrecía una visión en 3D más realista.

[1] http://es.wikipedia.org/wiki/Wiimote

(a)

Figura A.1: Gafas creadas para movernos.



(a)

Figura A.2: Escena en 3D usando OpenGL.

Bibliografía

[Alvarez Rey, 2001] Jose María Alvarez Rey. Implementación basada en lógica borrosa de jugadores para la robocup. *Proyecto Fin de Carrera. ITIS - Universidad Rey Juan Carlos*, 2001.

[Calvo Palomino, 2008] Roberto Calvo Palomino. Reconstrucción 3d mediante un sistema de atención visual. *Proyecto Fin de Carrera. Ing. Informática - Universidad Rey Juan Carlos*, 2008.

[Cañas and Matellán, 2002] José M. Cañas and Vicente Matellán. Dynamic gridmaps: comparing building techniques. *Universidad Rey Juan Carlos*, Abril 2002.

[Cañas *et al.*, 2009] José M. Cañas, Domenec Puig, Eduardo Perdices, and Tomás González. Visual goal detection for the robocup standard platform league. *Workshop of Physical Agents*, 2009.

[Chung Lee, 2008] Johnny Chung Lee. Head tracking for desktop vr displays using the wii remote. *Wii Remote Projects*, 2008.

[Crespo, 2003] María Ángeles Crespo. Localización probabilística en un robot con visión local. *Proyecto Fin de Carrera. ITIS - Universidad Rey Juan Carlos*, 2003.

[Hidalgo Blázquez, 2006] Víctor Manuel Hidalgo Blázquez. Comportamiento de persecución de un congénere con el robot pioneer. *Proyecto Fin de Carrera. ITIS - Universidad Rey Juan Carlos*, 2006.

[Kachach, 2005] Redouane Kachach. Localización del robot pioneer basada en láser. *Proyecto Fin de Carrera. ITIS - Universidad Rey Juan Carlos*, 2005.

[Kachach, 2008] Redouane Kachach. Calibración automática de cámaras en la plataforma jdec. *Proyecto Fin de Carrera. Ing. Informática - Universidad Rey Juan Carlos*, 2008.

[Larusso *et al.*, 1995] A. Larusso, D.W. Eggert, and R.B. Fisher. A comparison of four algorithms for estimating 3-d rigid transformations. *British Machine Vision Conference*, 1995.

[López Fernández, 2005] Alberto López Fernández. Localización visual del robot pioneer. *Proyecto Fin de Carrera. ITIS - Universidad Rey Juan Carlos*, 2005.

[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.

[Martínez de la Casa, 2003] Marta Martínez de la Casa. Comportamiento sigue pelota con visión cenital. *Proyecto Fin de Carrera. ITIS - Universidad Rey Juan Carlos*, 2003.

[Muelas Sahagún, 2009] David Muelas Sahagún. Visualizador 3d interactivo para laboratorios de análisis de marcha. *Proyecto Fin de Carrera. ITIS - Universidad Rey Juan Carlos*, 2009.

[Peño, 2003] Manuel Peño. Comportamiento saque de banda para la robocup. *Proyecto Fin de Carrera. ITIS - Universidad Rey Juan Carlos*, 2003.

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

[Smith *et al.*, 2006] P. Smith, I. Reid, and Andrew J. Davison. Real-time monocular slam with straight lines. *British Machine Vision Conference*, Septiembre 2006.

[Vega Pérez, 2008] Julio Vega Pérez. Navegación y autolocalización de un robot guía de visitantes. *Proyecto Fin de Carrera. Ing. Informática - Universidad Rey Juan Carlos*, 2008.