

Robots y sistemas autónomos
Máster en Automática, Robótica y Telemática

Curso 2009-2010

*Herramientas de simulación robótica. Construcción de un simulador para
Quadrotors.*

A. David Pinelo Moruno

Abstract – El presente trabajo de curso realiza un análisis preliminar de herramientas de simulación robóticas en software libre. De ellas, y eligiendo Gazebo como sistema base, se abordará los pasos necesarios para la construcción de un entorno multirobot de simulación para quadrotors (UAV). Se estudiará la arquitectura software adecuada, así como los modelos físicos que definen la dinámica y cinemática del sistema y su implementación en el simulador.

Índice de contenido

1. Introducción.....	4
2. Componentes software de un simulador.....	4
2.1. Motores de simulación física.....	5
2.1.1. Open Dynamics Engine.....	5
2.1.2. Bullets Physics.....	7
2.2. Representación 3D.....	7
2.2.1. OpenGL.....	9
2.2.2. Ogre 3D.....	9
2.3. Herramientas de simulación robótica existentes en software libre.....	10
2.3.1. Gazebo.....	10
3. Modelo físico del quadrotor.....	11
4. Implementación.....	12
4.1. Desarrollo de un controlador para Gazebo.....	13
4.2. Definición de un mundo virtual.....	14
4.2.1. Construcción física.....	14
4.2.2. Seleccionando el Mesh.....	15
4.2.3. Cambiando el material.....	15
4.2.4. Creando uniones o articulaciones.....	15
4.2.5. Cuerpos con múltiples geometrías.....	15
4.2.6. Controladores.....	15
4.2.7. XML Include.....	16
4.2.8. Nesting Models.....	16
4.3. Creación de una aplicación que interaccione con el simulador.....	16
4.3.1. Comunicación mediante libgazebo.....	17
4.3.2. Desarrollo de controladores para el control automático del vuelo.....	17
4.3.3. Interfaz de usuario.....	17
5. Conclusiones.....	18
6. Referencias.....	19

1. Introducción

El objetivo del presente trabajo de asignatura de Máster es el estudio e implementación de una herramienta de simulación robótica para quadrotors. La estructura de la presente memoria será la siguiente

1. Se *analizarán los componentes software necesarios para la implementación*. Estos contendrán entre otros la necesidad de un motor que simule la física del mundo real, la representación 3D del mundo a simular, o el motor de configuración que permite definir mundos virtuales de simulación.
2. Se estudiará el *modelo físico* del quadrotor, que habrá de implementarse en el simulador. Se analizarán las variables de estado y las variables de entrada que definirán el comportamiento del sistema. El modelo dinámico será el principal objeto de estudio.
3. Se presentará el proceso de *implementación* del modelo físico en la herramienta de simulación escogida.
4. Se presentarán diferentes *controladores* que pueden implementarse para *estabilizar el vuelo* del quadrotor, y que serán la base para algoritmos de planificación de trayectorias, cooperativos u otros que serán los que se simulen en esta herramienta.
5. Se afrontará el desarrollo de una *interfaz de comunicación* con el simulador que sirva de base para explicar cómo es la interacción con el simulador de las diferentes sistemas que quieran simularse.

La construcción de cualquier simulador es una labor muy compleja, y generalmente realizada por un equipo multidisciplinar capaz de implementar todas las particularidades interesantes que deben modelarse. Es por ello, que el presente trabajo no pretende realizar un simulador completo, sino más bien, mostrar el camino adecuado para una correcta implementación. Para ello, se mostrarán algunos resultados que serán de utilidad.

Por otro lado, se intentará minimizar en la medida de lo posible la presencia de código. El código fuente de la aplicación se encuentra debidamente comentado, por lo que en esta memoria se presentará líneas generales de implementación así como características de herramientas para determinar el porqué de su elección.

El simulador, basado en Gazebo¹ que se obtendrá permitirá

- *Simular el vuelo de múltiples quadrotors* en una misma simulación
- Permitirá interactuar con otros robots (terrestres) lo que lo hará adecuado para *entornos multirrobots*.
- *Configuración sencilla* y versátil de los diferentes mundos de simulación
- Posibilidad de *añadir* al quadrotor *cargas*, u otros objetos o sensores.

2. Componentes software de un simulador

La construcción de un simulador de cualquier tipo implica el desarrollo e interacción de diferentes componentes software. En esta memoria no se presentarán todos los componentes, sólo aquellos que aportan interés desde un punto de vista de la robótica.

Por otro lado, no se establecerá ningún software específico como punto de partida. Se presentarán componentes software que de por sí, podrían integrarse para realizar el simulador. Una vez estudiados, se pondrá de manifiesto que las herramientas de simulación robótica actuales utilizan estos mismos componentes, de tal forma que será una de las existentes la escogida.

¹Simulador en 3D, parte del proyecto Player/Stage.

Los componentes estudiados en esta memoria serán

- El *motor que simule un entorno físico*, realizando el correspondiente análisis y balance de fuerzas (a partir de la mecánica newtoniana), velocidades, aceleraciones, entre otras. Será también el encargado de detectar colisiones o modelar carga en sistemas, entre otros.
- El *motor de representación 3D* para visualizar los diferentes objetos que quieren simularse.
- *Entorno de configuración* que permita crear o modificar mundos virtuales simulados, de manera rápida y ágil.

2.1. Motores de simulación física

Los motores de simulación física son componentes software o tipo de programación, que supone la introducción de las leyes de Física en un simulador o motor de juego, particularmente en aquellos sistemas de información que tienen en los gráficos 3D su aspecto determinando. El propósito es hacer que los efectos físicos de los objetos creados o modelados tengan las mismas características que en la vida real, teniendo en cuenta, por ejemplo, gravedad, masa o fricción, por ejemplo. El Motor físico, por tanto, es una API de programación usada para simular la mecánica newtoniana en el ambiente.

La simulación de la física en la programación es solo una aproximación cercana a la física real (si bien se acerca mucho a la física que tendría un objeto en realidad, no es igual de realista que en la realidad), y el cálculo es desarrollado usando valores discretos. Hay muchos elementos que forman los componentes de la simulación física. Entre los más utilizados

- El *detector de colisiones*, que se utiliza para resolver el problema de determinar cuándo es que dos o más objetos físicos en el ambiente se cruzan entre sí.
- El *motor físico* en sí, usado para implementar las leyes de la mecánica newtoniana.

Se presentarán dos motores físicos en software libre, cuyas características se enuncian.

2.1.1. Open Dynamics Engine

La API Open Dynamics Engine (ODE) es una librería libre y de calidad industrial para simular la dinámica de cuerpos rígidos y articulados. Por ejemplo, es muy adecuada para simular vehículos de tierra o mover objetos en un ambiente de realidad virtual. Es rápida, flexible y robusta. Implementa además detección de colisiones. El desarrollo de ODE lo inicia Russell Smith y cuanta con la colaboración de diversas fuentes.

ODE es muy adecuada para simular estructuras rígidas articuladas. Una estructura articulada se crea cuando se conectan diferentes cuerpos rígidos con uniones de diferentes tipos. Como ejemplo, se encuentran los vehículos terrestres (donde las ruedas se conectan al chasis).

Está diseñada para ser utilizada en *simulaciones de tiempo real*. Es particularmente adecuada para simular objetos que se mueven en un ambiente de realidad virtual, ya que es rápida, robusta y estable, y el usuario tiene la completa libertad para cambiar la estructura del sistema mientras la simulación está corriendo.

ODE utiliza un *integrador* altamente estable, de tal manera que los errores de simulación no crezcan fuera de control. El sentido físico de esto que el sistema simulado no “explota” por ninguna razón, algo muy común en otras simulaciones. ODE hace mucho énfasis en la velocidad y la estabilidad sobre la mera certeza física.

Implementa además contactos fuertes: es decir, se utiliza una restricción no-penetrante cuando dos cuerpos colisionan.

Implementa también un sistema de *detección de colisiones*. Pero, puede implementarse cualquier sistema de detección de colisiones adicional. Las primitivas de colisiones actualmente implementadas son esferas, cajas, cilindros, planos, rayos, aunque pueden implementarse otras más.

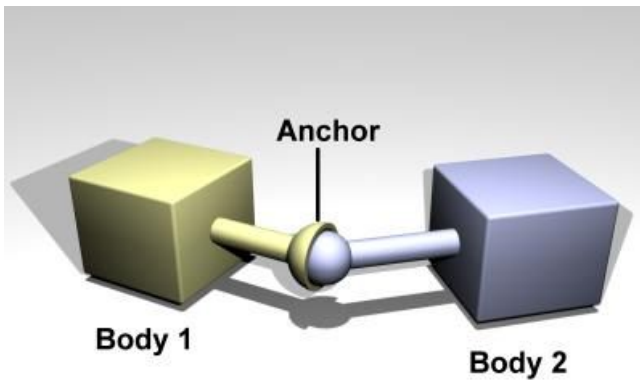


Ilustración 2: Ball and socket

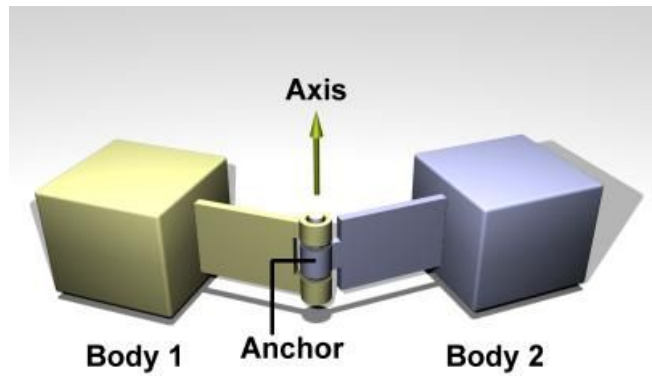


Ilustración 3: Hinge

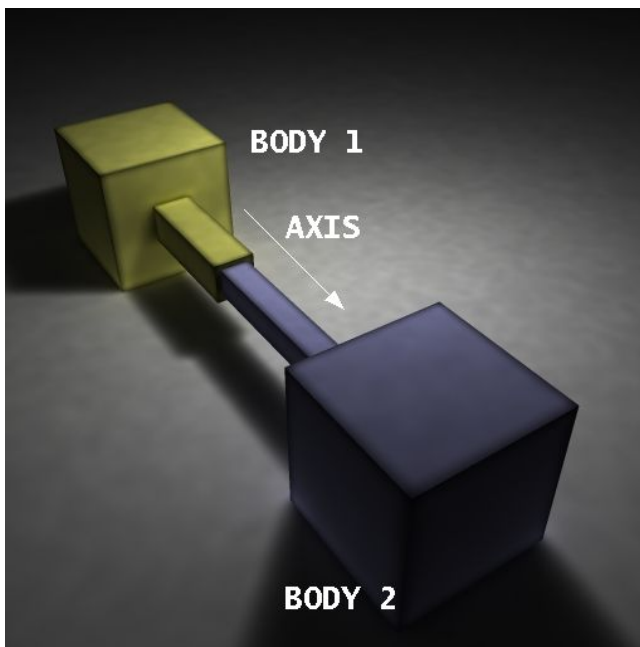


Ilustración 5: Slider

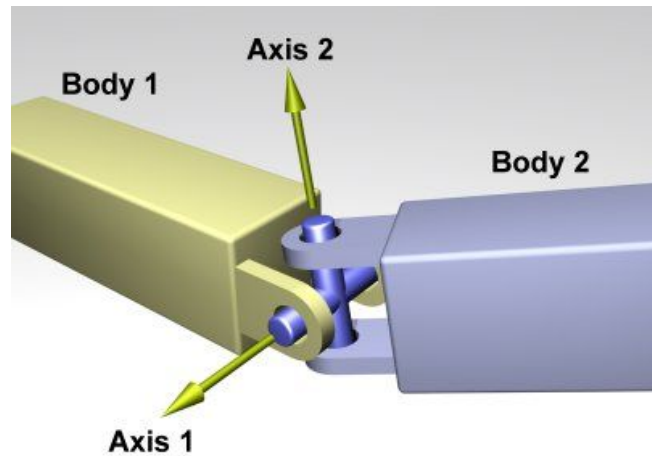


Ilustración 4: Universal

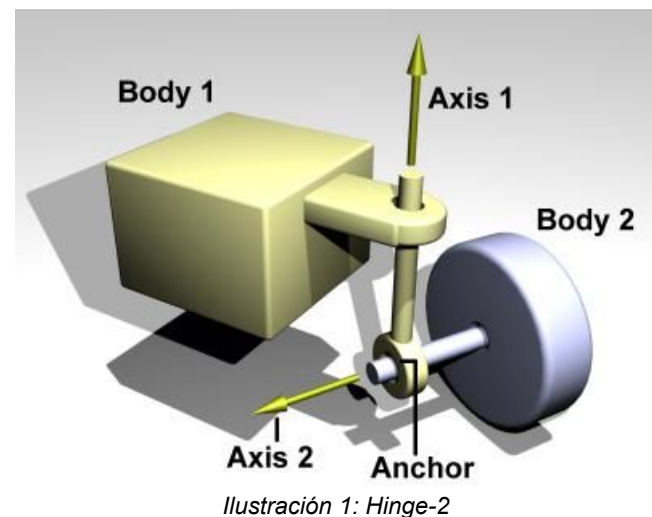


Ilustración 1: Hinge-2

Algunas características

- Cuerpos rígidos con una distribución arbitraria de masas.
- Diferentes tipos de articulaciones:
 - Ball-and-socket

- Hinge
 - Slider (prismatic)
 - Hinge-2
 - Fija
 - Universal
- Primitivas de colisión: Esferas, cajas, cilindros, planos, rayos y mallas triangulares
 - Espacios de colisiones: Quad tree, hash space y simple
 - Métodos de simulación: Las ecuaciones de movimiento se derivan del modelo de velocidad basado en multiplicadores de Lagrange gracias a Trinkle/Stewart y Anitescu/Potra.
 - Se utiliza un integrador de primer orden. Es muy rápido aunque no proporciona resultados del todo adecuado para entornos más susceptibles
 - Elección del tiempo de simulado
 - Modelos de contactos y fricción: Basados en el Dantzig LCP solver descrito por Baraff, aunque ODE implementa una aproximación más rápida al modelo de fricción de Coloumb
 - Tiene una interfaz nativa en C (aunque ODE está escrito casi entero en C++)
 - Tiene una interfaz escrita en C++ sobre la interfaz de .
 - Incorpora muchos test unitarios
 - Tiene optimizaciones específicas para cada plataforma

2.1.2. Bullets Physics

Bullet Physics es una librería profesional en software libre para la detección de colisiones y simulación de la dinámica de cuerpos rígidos y blandos. Esta librería es libre para uso comercial bajo la licencia ZLib.

Entre sus características principales:

- Código en C++ disponible para PLAYSTATION 3, XBox 360, Wii, PC, Linux, Mac OSX e iPhone
- Detección continua y discreta de colisiones
- Solución de restricciones en la dinámica de cuerpos rígidos rápida y estable
- Dinámica para ropas y volúmenes deformables, en interacción con cuerpos rígidos.
- Plugin para la integración con Maya o Blender.

2.2. Representación 3D

La creación de gráficos 3D es un proceso complejo, desarrollado en los últimos años gracias al incremento de rendimiento de los diferentes sistemas informáticos. Es interesante mostrar las fases para la creación de elementos/gráficos 3D:

- *Modelado*. La etapa de modelado consiste en ir dando forma a objetos individuales que luego serán usados en la escena.

- *Sombreado/Texturizado*. Definición de la forma que afecta la luz a los diferentes elementos representados. Se utiliza para ello materiales «*shaders*» que son algoritmos que controlan la incidencia de la luz, produciendo materiales de tipo: Anisótropo, Lambert, Blin... Es posible también combinarlas con texturas.
- *Iluminación*. Creación de luces de diversos tipos puntuales, direccionales en área o volumen, con distinto color o propiedades.
- *Animación*. Los objetos se pueden animar en cuanto a
 - Transformaciones básicas en los tres ejes (XYZ), Rotación, Escala o Traslación.
 - Forma(shape):
 - Mediante esqueletos: a los objetos se les puede asignar un esqueleto, una estructura central con la capacidad de afectar la forma y movimientos de ese objeto. Esto ayuda al proceso de animación, en el cual el movimiento del esqueleto automáticamente afectara las porciones correspondientes del modelo.
 - Mediante deformadores: ya sean lattices (cajas de deformación) o cualquier deformador que produzca por ejemplo deformación sinusoidales.
 - Dinámicas: para simulaciones de ropa, pelo, dinámicas rígidas de objeto.
- *Renderizado*. Se llama rénder al proceso final de generar la imagen 2D o animación a partir de la escena creada. Esto puede ser comparado a tomar una foto o en el caso de la animación, a filmar una escena de la vida real. Generalmente se buscan imágenes de calidad fotorrealista, y para este fin se han desarrollado muchos métodos especiales. Las técnicas van desde las más sencillas, como el rénder de alambre (wireframe rendering), pasando por el rénder basado en polígonos, hasta las técnicas más modernas como el Scanline Rendering, el Raytracing, la radiosidad o el Mapeado de fotones
 - El software de rénder puede simular efectos cinematográficos como el lens flare, la profundidad de campo, o el motion blur (desenfoco de movimiento). Estos artefactos son, en realidad, un producto de las imperfecciones mecánicas de la fotografía física, pero como el ojo humano está acostumbrado a su presencia, la simulación de dichos efectos aportan un elemento de realismo a la escena. Se han desarrollado técnicas con el propósito de simular otros efectos de origen natural, como la interacción de la luz con la atmósfera o el humo. Ejemplos de estas técnicas incluyen los sistemas de partículas que pueden simular lluvia, humo o fuego, el muestreo volumétrico para simular niebla, polvo y otros efectos atmosféricos, y las cáusticas para simular el efecto de la luz al atravesar superficies refractantes. Algunos de estos efectos están presentes en el simulador a desarrollar.
 - El proceso de rénder necesita una gran capacidad de cálculo, pues requiere simular gran cantidad de procesos físicos complejos. La capacidad de cálculo se ha incrementado rápidamente a través de los años, permitiendo un grado superior de realismo en los rénders.

Los gráficos 3D se han convertido una técnica muy utilizada, particularmente en juegos y herramientas de simulación. Se han creado APIs especializadas para facilitar los procesos en todas las etapas de la generación de gráficos 3D. Estas APIs han demostrado ser vitales para los desarrolladores de hardware para gráficos informáticos, ya que proveen un camino al programador para acceder al hardware

de manera abstracta, aprovechando las ventajas de la placa de vídeo. Las siguientes APIs para gráficos por computadora son particularmente populares:

- OpenGL
- Direct3D (subconjunto de DirectX).

2.2.1. OpenGL

OpenGL (Open Graphics Library) es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D. La interfaz consiste en más de 250 funciones diferentes que pueden usarse para dibujar escenas tridimensionales complejas a partir de primitivas geométricas simples, tales como puntos, líneas y triángulos. Fue desarrollada originalmente por Silicon Graphics Inc. (SGI) en 1992 y se usa ampliamente en CAD, realidad virtual, representación científica, visualización de información y simulación de vuelo. También se usa en desarrollo de videojuegos, donde compite con Direct3D en plataformas Microsoft Windows.

Fundamentalmente OpenGL es una *especificación*, es decir, un documento que describe un conjunto de funciones y el comportamiento exacto que deben tener. Partiendo de ella, los fabricantes de hardware crean implementaciones, que son bibliotecas de funciones que se ajustan a los requisitos de la especificación, utilizando aceleración hardware cuando es posible. Dichas implementaciones deben superar unos tests de conformidad para que sus fabricantes puedan calificar su implementación como conforme a OpenGL y para poder usar el logotipo oficial de OpenGL.

OpenGL tiene dos propósitos esenciales:

- *Ocultar la complejidad de la interfaz con las diferentes tarjetas gráficas*, presentando al programador una API única y uniforme.
- *Ocultar las diferentes capacidades de las diversas plataformas hardware*, requiriendo que todas las implementaciones soporten la funcionalidad completa de OpenGL (utilizando emulación software si fuese necesario).

El funcionamiento básico de OpenGL consiste en aceptar primitivas tales como puntos, líneas y polígonos, y convertirlas en píxeles. Este proceso es realizado por una pipeline gráfica conocida como la Máquina de estados de OpenGL. La mayor parte de los comandos de OpenGL o bien emiten primitivas a la pipeline gráfica o bien configuran cómo la pipeline procesa dichas primitivas.

OpenGL es una API basada en procedimientos de bajo nivel que requiere que el programador dicte los pasos exactos necesarios para renderizar una escena. Esto contrasta con las APIs descriptivas, donde un programador sólo debe describir la escena y puede dejar que la biblioteca controle los detalles para representarla. El diseño de bajo nivel de OpenGL requiere que los programadores conozcan en profundidad la pipeline gráfica, a cambio de darles libertad para implementar algoritmos gráficos novedosos.

Según esto último, se suele utilizar librerías como Ogre para simplificar la escritura de código.

2.2.2. Ogre 3D

OGRE 3D (acrónimo del inglés Object-Oriented Graphics Rendering Engine) es un motor de renderizado 3D orientado a escenas, escrito en el lenguaje de programación C++.

Sus bibliotecas evitan la dificultad de la utilización de capas inferiores de librerías gráficas como OpenGL y Direct3D, y además, proveen una interfaz basada en objetos del mundo y otras clases de alto nivel. El motor es software libre, licenciado bajo MIT y con una comunidad muy activa.

Características destacables

- Interfaz orientada a objeto de fácil uso que minimiza el esfuerzo para renderizar escenas 3D, y que intenta ser independiente de cualquier implementación 3D como Direct3D u OpenGL
- Framework extensible que permite codificar la aplicación de forma rápida y sencilla

- Así por ejemplo, los requisitos comunes para realizar el *réndér* (como transparencias por ejemplo) se realizan automáticamente
- Documentación clara y extensa de todas las clases del motor
- Probado y estable. Usado en diferentes productos comerciales
- Soporta Direct3D y OpenGL
- Presente en Windows, Linux y Mac OSX
- Soporte para múltiples técnicas de material que permiten diseñar efectos alternativos muy adecuados.
- Carga de texturas desde archivos PNG, JPEG, TGA, BMP o DDS.
- Soporte para animaciones muy sofisticadas a partir de esqueletos

2.3. Herramientas de simulación robótica existentes en software libre

Un simulador robótico se utiliza para crear aplicaciones empujadas para un robot sin depender físicamente de la maquinaria, permitiendo así ahorrar costes y tiempo. En algunos casos, estas aplicaciones pueden transferirse o portarse al robot real sin modificaciones. El término simulador robótico, puede referirse a diferentes aplicaciones de simulación del ámbito de la robótica. Por ejemplo, en robots móviles, los simuladores basados en comportamiento permite a los usuarios el crear mundos sencillos de objetos y fuentes de luz y programar robots para interactuar con estos mundos.

Los simuladores modernos suelen presentar las siguientes características

- Definición rápida de prototipos de robots
 - Usando el propio simulador como herramienta de creación (Webots, R-Station, Marilou).
 - Usando herramientas externas (Gazebo usa Blender).
- Motores de física para simular movimientos realistas. La mayoría de simuladores utilizan ODE (Gazebo, LpzRobots, Marilou, Webots) o PhysX (Microsoft).
- Renderizado realista en 3D. Las herramientas estándar de modelado en 3D se utilizan para construir los diferentes mundos o entornos.
- Utiliza motores de script para controlar la dinámica de los robots: URBI, MATLAB, o Python son los lenguajes usados por Webots. Gazebo utiliza Python.

2.3.1. Gazebo

Gazebo es un simulador robótico en 3D que forma parte del Proyecto Player.

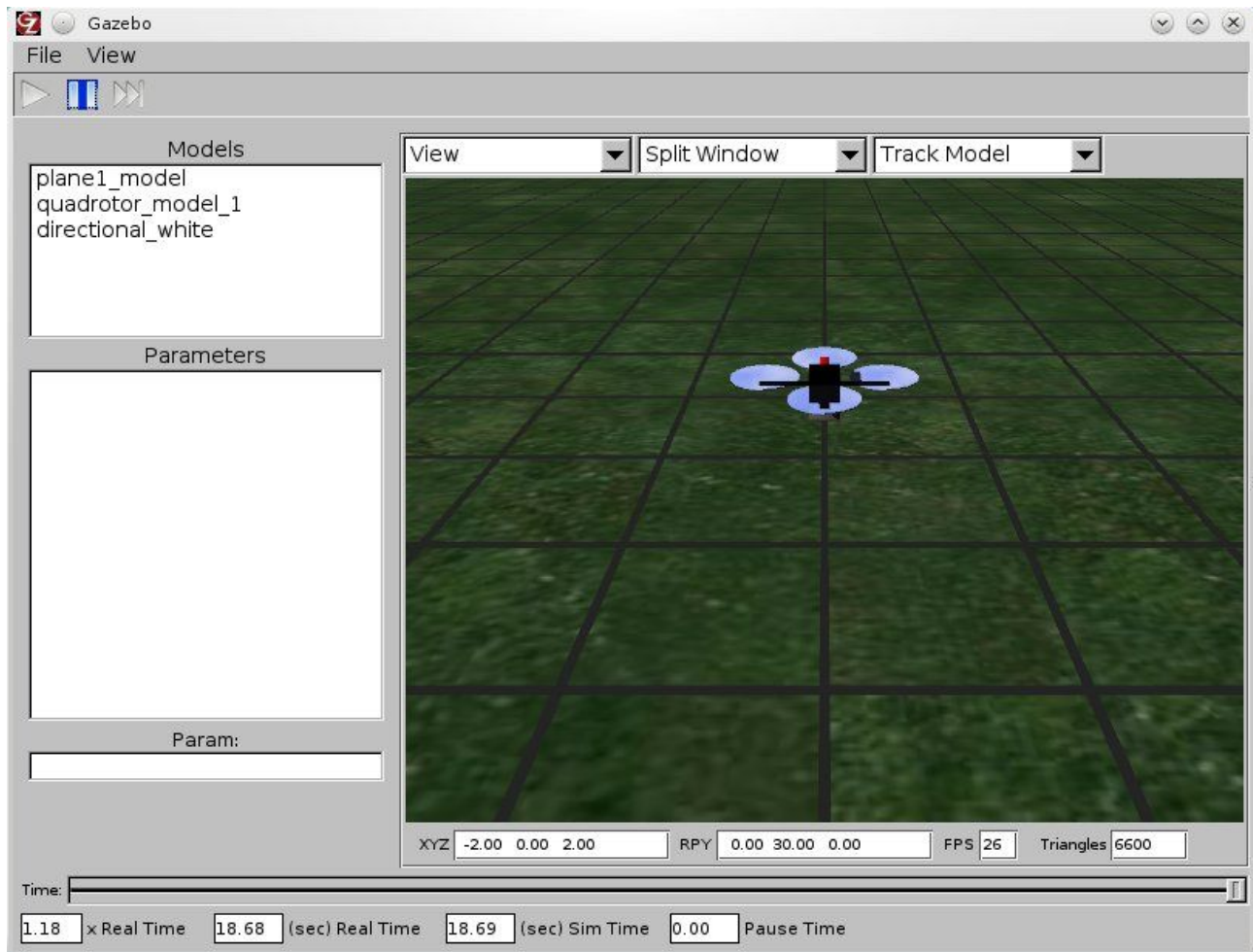
El Proyecto Player es un proyecto para crear software libre para investigar en sistemas robóticos y sensoriales. Sus componentes incluyen el servidor de red Player y las plataformas de simulación Gazebo y Stage. Player es probablemente la interfaz robótica más utilizada en investigaciones.

Gazebo es un simulador multirrobot para diferentes entornos (indoor y outdoor). Es capaz de simular una población de robots, sensores y objetos en un mundo tridimensional. Genera un feedback realista en los sensores e interacciones plausibles entre objetos (incluye para ello una adecuada simulación de la física de cuerpos rígidos, basado en ODE).

Normalmente, se suele utilizar Gazebo junto con Player, que actúa como un mecanismo abstracto a modo de servidor a través del cual los diferentes controladores de robots (los clientes) interactúan con el hardware del robot. Player proporciona así datos simulados para estos controladores

en lugar de lecturas reales de sensores. En principio, los controladores no notan la diferencia entre dispositivos reales y la simulación de Gazebo de esos dispositivos.

Gazebo puede ser también controlado a través de una API de bajo nivel escrita en C (*libgazebo*) y C++. Esta librería permite incluir desarrollos que se integran con Gazebo en su propia arquitectura. Este será el enfoque utilizado para este proyecto. Libgazebo permite por tanto interactuar directamente con las variables que controlan la simulación, y será por tanto la interfaz adecuada sobre la que crear las diferentes aplicaciones que quieran simularse.



3. Modelo físico del quadrotor

Se recurrirá a la tesis doctoral de *Samir Bouabdallah* [1] para modelar dinámicamente el vuelo de un quadrotor. Se presentará directamente el modelo dinámico deducido

$$\ddot{x} = (\cos \phi \operatorname{sen} \theta \cos \psi + \sin \phi \sin \psi) \frac{1}{m} U_1$$

$$\ddot{y} = (\cos \phi \operatorname{sen} \theta \sin \psi - \sin \phi \cos \psi) \frac{1}{m} U_1$$

$$\ddot{z} = -g + (\cos \phi \cos \theta) \frac{1}{m} U_1$$

$$\ddot{\phi} = \dot{\theta} \dot{\psi} \left(\frac{I_y - I_z}{I_x} \right) - \frac{J_r}{I_x} \dot{\theta} \Omega + \frac{l}{I_x} U_2$$

$$\ddot{\theta} = \dot{\phi} \dot{\psi} \left(\frac{I_z - I_x}{I_y} \right) + \frac{J_r}{I_y} \dot{\phi} \Omega + \frac{l}{I_y} U_3$$

$$\ddot{\psi} = \dot{\theta} \dot{\phi} \left(\frac{I_x - I_y}{I_z} \right) + \frac{1}{I_z} U_4$$

donde

$$U_1 = b(\Omega_1^2 + \Omega_2^2 + \Omega_3^2 + \Omega_4^2)$$

$$U_2 = b(\Omega_4^2 - \Omega_2^2)$$

$$U_3 = b(\Omega_3^2 - \Omega_1^2)$$

$$U_4 = b(\Omega_2^2 + \Omega_4^2 - \Omega_1^2 - \Omega_3^2)$$

$$\Omega = \Omega_2 + \Omega_4 - \Omega_1 - \Omega_3$$

Las variables de entrada al sistema serán las velocidades angulares de los cuatro rotores del quadrotor, $\Omega_1, \Omega_2, \Omega_3, \Omega_4$.

El estado del sistema estará descrito por doce variables.

- Las *posiciones* del quadrotor en el espacio cartesiano, x, y, z
- Las *velocidades lineales* del quadrotor en los ejes x, y, z
- Las *aceleraciones* en los ejes x, y, z
- La *orientación* del quadrotor, roll, pitch y yaw, θ, ϕ, ψ
- Las *velocidades de rotación*
- Las *aceleraciones en la rotación*

Por tanto, el sistema de simulación recibirá como entradas, las velocidades de giro de los cuatro rotores, y presentará en la simulación la evolución de las doce variables anteriores.

Es interesante destacar que el modelo dinámico del sistema es un balance de fuerzas newtonianas. Es en este punto donde se produce la integración con el motor físico visto en apartados anteriores.

4. Implementación

Se describirá en este apartado el proceso de construcción del simulador en Gazebo. Para ello, son necesarios varios pasos.

1. Desarrollo de un *controlador* para el quadrotor. Este controlador, en terminología Gazebo será el elemento utilizado para simular la dinámica del quadrotor. Será en el controlador donde se implementarán las ecuaciones de balance de fuerzas anteriormente presentados. El controlador a su vez, será el encargado de recibir las variables de entrada (velocidades de giro de los rotores) para aplicarlas al objeto de simulación.
2. Desarrollo de un *mundo virtual* en el que simular.

3. Desarrollo de una interfaz de usuario que, utilizando libgazebo, interactuará con el simulador. Leerá los valores de usuario en cuanto a valores de navegación que el quadrotor debe tomar (roll, pitch, yaw o altura). Esta interfaz incorporará también controladores (PID, Lyapunov, LQR) que calcularán adecuadamente, mediante bucle cerrado, los valores adecuados de las velocidades de giro de los rotores para alcanzar los valores de roll, pitch, yaw o altura indicados.

Las versiones utilizadas para el desarrollo:

- Gazebo: 0.10.0
- ODE: 0.11.1
- OGRE: 1.6.4
- Qt: 4.6.2

4.1. Desarrollo de un controlador para Gazebo

Para simular el quadrotor dentro de Gazebo, es necesario definir en algún punto del mismo, la dinámica del UAV. Es necesario indicar a Gazebo cuáles serán las variables de entradas del quadrotor y cuál será el comportamiento del mismo ante esas variables de entrada.

Esto se realiza en lo que Gazebo denomina *controlador*. Los controladores se utilizan para salvar la distancia entre el interfaz de programación con Gazebo (libgazebo) y los atributos físicos del modelo. Por tanto el controlador, implementa el modelo dinámico y entre sus funciones se encuentran

- Establecer el movimiento adecuado de las articulaciones (en robots articulados)
- Escribir comandos a libgazebo (como son las velocidades angulares de los rotores, Ω_i)
- Leer comandos de libgazebo

A partir de los valores de las velocidades de giro de los rotores, Ω_i , es posible determinar las fuerzas aplicadas al quadrotor. Para ello, y examinando el modelo dinámico del quadrotor anteriormente presentado, es fácil deducir que corresponden a ecuaciones de balance de fuerzas por cada dimensión lineal, y balance de momentos aplicados para determinar la orientación del quadrotor en el espacio.

El controlador en Gazebo deriva de la clase `Controller` que implementa una interfaz necesaria para la utilización por parte de Gazebo. No se entrará en explicar con detalle esta interfaz ya que se encuentra adecuadamente explicada en la documentación de la herramienta[2], aunque sí se comentará brevemente algunas particularidades.

1. Debe implementar una función, `LoadChild` que leerá la configuración que el controlador necesita. Lo hará a partir de un fichero XML que le será pasado a través del archivo de mundo adecuado. Entre esta configuración, y para este trabajo, se encontrarán variables como la masa del quadrotor, sus inercias, y variables de vuelo como los factores de thrust o drag.
2. Debe implementar funciones de inicialización del controlador (donde se establecen condiciones iniciales del quadrotor), y de finalización del mismo (liberación de memoria entre otras). Estas funciones son `InitChild` y `FiniChild`.
3. Debe implementar la función `UpdateChild`, encargada de actualizar para cada cambio de las variables de entrada, el estado del quadrotor. Esta función será invocada cada vez que las velocidades angulares de los rotores cambien, ajustando las nuevas fuerzas que se ejercen sobre el UAV.

Por tanto, la función `UpdateChild` será la función donde se implementará el balance de fuerzas. De tal manera, que un cambio en las velocidades Ω_i , provocará unas nuevas fuerzas que serán calculadas y aplicadas al cuerpo del quadrotor en la simulación, mediante la correspondiente interfaz con ODE que proporciona Gazebo.

4.2. Definición de un mundo virtual

El controlador definido en el apartado anterior se utiliza en una determinada simulación, definida por un mundo. Los mundos en Gazebo, son archivos en XML que definen cómo será el ambiente de realidad virtual en el que se simulará.

Se presenta en este apartado una breve guía de cómo se construye un archivo mundo en Gazebo.

La definición de un mundo en Gazebo consta de diferentes partes o secciones. Un mundo contiene diferentes objetos que interactúan entre sí. Para definir estos objetos o modelos, se pueden definir los siguientes aspectos

- *Construcción física*: Creación básica del modelo.
- *Definición de Mesh*: Elección de la "piel" o skin del modelo
- *Definición del material*: Se selecciona el modelo de material que presenta el modelo.
- *Creación de las articulaciones* o uniones
- *Se crea el cuerpo con múltiples geometrías*
- *Controladores*: Se añaden los controladores necesarios al modelo (si el modelo es el quadrotor, se agregará el controlador definido anteriormente).
- *XML Include*: Es adecuado definir modelos diferentes en diferentes archivos. LA inclusión de ficheros permite la limpieza de la definición del mundo.
- *Nesting Models*: Permite la conexión de modelos en una relación padre-hijo.

Los modelos, forma por tanto, el núcleo de la simulación en Gazebo. Aglutinan todas las entidades físicas y sensores que los definen y darán forma a la simulación.

4.2.1. Construcción física

Todos los aspectos físicos de un modelo se definen en un archivo XML. Incluye posición, orientación, uniones o articulaciones, densidad. Se deben tener en cuenta algunas reglas básicas para la construcción de un modelo:

- Dar un nombre adecuado a todo: El nombre debe ser único.
- Un modelo debe contener un cuerpo, *body*.
- Un cuerpo puede contener sólo sensores y geometrías.

Puede mostrarse un ejemplo, creando como modelo una simple caja:

```
<model:physical name="box_model">
  <xyz>0 1.5 0.5</xyz>
  <rpy>0.0 0.0 0.0</rpy>
  <canonicalBody>box1_body</canonicalBody>
  <static>>false</static>

  <body:box name="box_body">
    <xyz>0.0 0.0 0.0</xyz>
    <rpy>0.0 0.0 0.0</rpy>

    <geom:box name="box_geom">
      <xyz>0.0 0.0 0.0</xyz>
      <rpy>0.0 0.0 0.0</rpy>
      <mesh>default</mesh>
      <size>1 1 1</size>
```

```

    <density>1.0</density>
    <material>Gazebo/BumpyMetal</material>
  </geom:box>
</body:box>
</model:physical>

```

En este ejemplo, la posición de inicio se sitúa en 1.5 metros en el eje Z, y 0.5 metros en el eje Y. El modelo no tiene rotación.

El tag `canonicalBody` define a qué cuerpo dentro del modelo debe anidarse el modelo que se está definiendo. El anidamiento ocurre cuando un modelo es definido dentro de otro modelo. Se crea así una relación padre a hijo y una unión agregada a los dos modelos que quedan conectados por sus cuerpos canónicos respectivos.

El flag `static` indica si el modelo está afectado o no por la física. Puesto a `false`, el modelo nunca se moverá, pero todavía podrá colisionar con otros objetos en movimiento. Si se establece a `true`, el motor de física, actualizará el modelo acorde a su masa, posición, colisiones...

El tag `size` indica cómo de grande será el modelo. El tag de `density` se usa para calcular la masa acorde al tamaño. Por último, el tag `material` indica qué textura se aplicará a la geometría.

4.2.2. Seleccionando el Mesh

Cambia la piel, *skin*, o aspecto de una geometría. El valor seleccionado debe coincidir con alguno de los modelos de OGRE. La definición se realiza con la siguiente línea

```
<mesh>fish</mesh>
```

4.2.3. Cambiando el material

El material que se aplica a una geometría se puede también cambiar a través de OGRE. La definición en XML

```
<material>Gazebo/FlatBlack</material>
```

4.2.4. Creando uniones o articulaciones

Las uniones conectan dos cuerpos de forma conjunta. Son útiles para conectar ruedas a un coche, o crear brazos articulados. Los tipos de uniones se definieron en el apartado en el que se hablaba de ODE.

Se encuentran buenos ejemplos de uniones en el modelo `Pionerer2dx` que incluye Gazebo.

4.2.5. Cuerpos con múltiples geometrías

Un cuerpo puede contener una o diferentes geometrías, lo que permite formas complejas. Cada geometría en un cuerpo debe tener una única posición y orientación relativa al cuerpo padre al que pertenece. Este ha sido la aproximación utilizada para crear la forma del quadrotor.

El archivo `quadrotor.model` presenta esta técnica para construir el modelo del quadrotor.

4.2.6. Controladores

Muchos modelos requieren poder moverse o publicar datos de un sensor implementado. Como ya se ha explicado, este se realiza mediante los controladores, al simular un dispositivo físico. Los controladores se compilan dentro de Gazebo y se utilizan o agregan al modelo utilizando XML. Para el caso del controlador definido para el quadrotor, esto se realiza mediante el siguiente código en XML.

```

<controller:quadrotor_force name="controller1">
  <bodyName>chassis_body</bodyName>
  <thrust_factor>3.13e-5</thrust_factor>
  <drag_factor>9e-7</drag_factor>
  <Ixx>0.0086</Ixx>
  <Iyy>0.0086</Iyy>
  <Izz>0.0172</Izz>
  <gravity>9.81</gravity>

```

```

    <mass>0.4794</mass>
    <Jr>3.7404e-5</Jr>
    <L>0.225</L>
    <interface:quadrotor name="quadrotor_iface"/>
  </controller:quadrotor_force>

```

Cada controlador, requiere su propio conjunto de parámetros, que dependerá de la funcionalidad implementada en el controlador. El parámetro `interface` permite enlazar el controlador con la interfaz de libgazebo. El nombre definido en esta interfaz será el utilizado en la aplicación externa que se desarrollará para interactuar con el simulador.

4.2.7. XML Include

Un modelo se puede definir en un archivo separado, e incluido en otro archivo o en el archivo de mundo. Se permite así tener la posibilidad de definir un único modelo y utilizarlo en tantos mundos como fuera necesario sin duplicar código. En este trabajo, se ha definido un modelo de quadrotor que podrá ser utilizado en tantas simulaciones como se desean.

Un modelo se puede incluir en un mundo con dos métodos. El primero asume que el modelo no requiere modificaciones, de tal forma que el código XML será

```

<include embedded="false">
  <xi:include href="quadrotor.model" />
</include>

```

Si el modelo a incluir presenta ligeras modificaciones, como un cambio de nombre o de posiciones, entonces debe usarse el siguiente método

```

<model:physical name="quadrotor_model1">
  <xyz>1 0 0.25</xyz>
  <rpy>0.0 0.0 0.0</rpy>

  <controller:quadrotor_force name="controller1">
    <bodyName>chassis_body</bodyName>
    <thrust_factor>3.13e-5</thrust_factor>
    <drag_factor>9e-7</drag_factor>
    <Ixx>0.0086</Ixx>
    <Iyy>0.0086</Iyy>
    <Izz>0.0172</Izz>
    <gravity>9.81</gravity>
    <mass>0.4794</mass>
    <Jr>3.7404e-5</Jr>
    <L>0.225</L>
    <interface:quadrotor name="quadrotor_iface"/>
  </controller:quadrotor_force>
<!--

```

El include debe ser lo último en el modelo. Todas las sentencias anteriores serán sobrescritas con este include.

```

-->
<include embedded="true">
  <xi:include href="quadrotor.model" />
</include>
</model:physical>

```

4.2.8. Nesting Models

Los modelos se pueden anidar, permitiendo así crear modelos complejos. Es especialmente útil cuando se agregan sensores a un robot, por ejemplo.

4.3. Creación de una aplicación que interactúe con el simulador

La aplicación a desarrollar presenta cuatro secciones diferenciadas.

- La comunicación con Gazebo a través de libgazebo
- El desarrollo de controladores (Lyapunov, PID, LQR) para controlar el vuelo del quadrotor
- La interfaz de usuario en sí

4.3.1. Comunicación mediante libgazebo

Se ha desarrollado una clase, `GazeboLink`, que provee de las herramientas necesarias para la comunicación con el simulador a través de la librería libgazebo. Principalmente esta clase provee

- Las primitivas de *conexión* y *desconexión* necesarias
- Las primitivas para la *escritura de datos*. Los datos a escribir son las velocidades de giro de cada rotor, Ω_i
- Las primitivas para la *lectura de datos*
 - Lectura de velocidades angulares de los robots actuales
 - Lectura de fuerzas y momentos aplicados al quadrotor
 - Lectura de posición y orientación, así como de velocidades y aceleraciones angulares y lineales, necesarias para cerrar el bucle
 - Lectura de secuencia de imágenes de una posible cámara adjuntada al quadrotor

El código está profusamente comentado por lo que no es necesario seguir incidiendo en el mismo en esta memoria.

4.3.2. Desarrollo de controladores para el control automático del vuelo

Seguiente la tesis de Samir Bouabdallah se han implementado tres controladores en bucle cerrado

- Control mediante el análisis de estabilidad del sistema de Lyapunov
- Control mediante un PID por cada rotor
- Control mediante LQR

El control se ha realizado en todos los casos mediante un timer, que cada intervalo de tiempo definido por el usuario lee las variables de estado del quadrotor, y determinar las señales de control a aplicar.

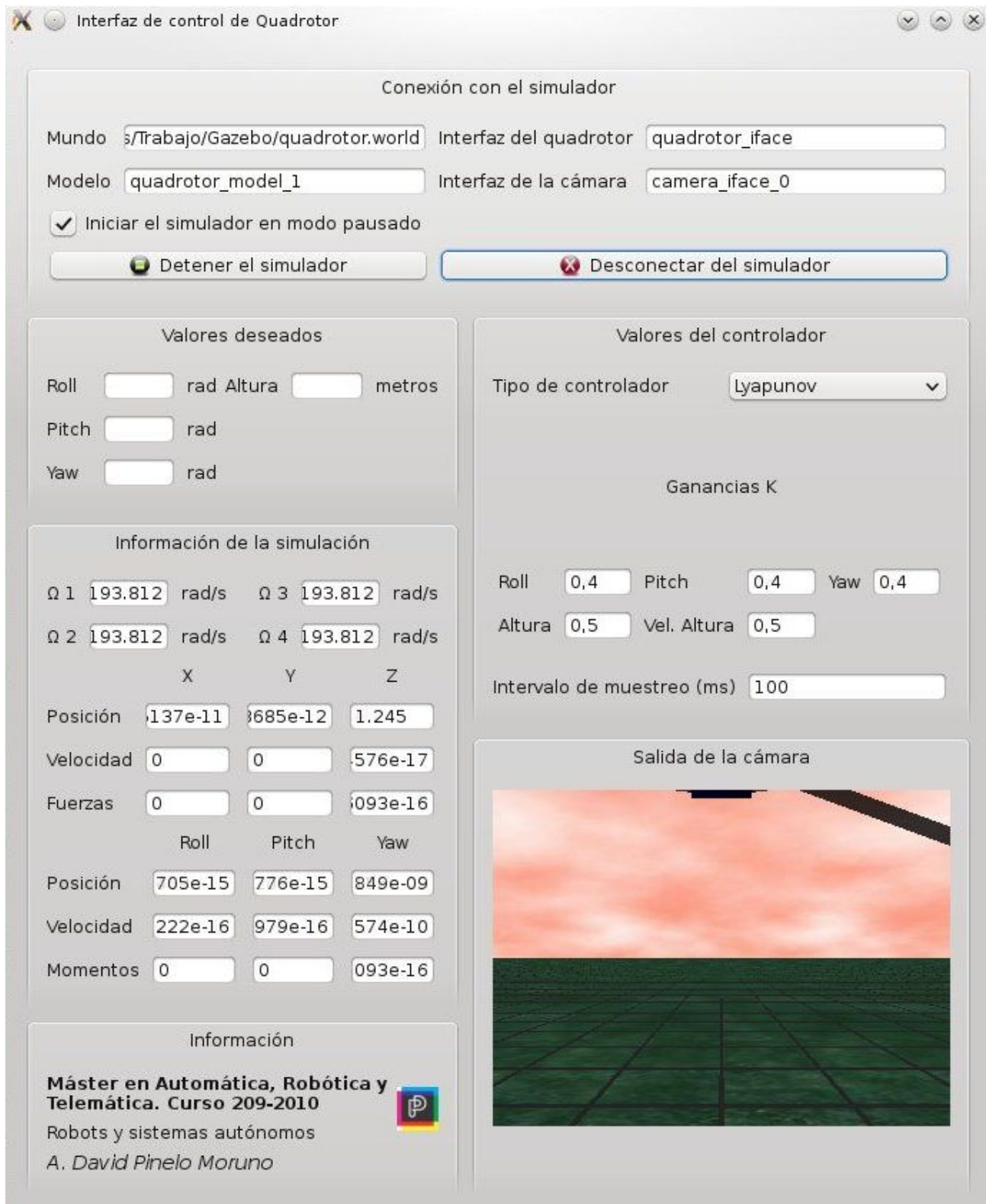
No se va a entrar en determinar el cálculo de los parámetros o constantes de los diferentes controladores, ya que es materia ampliamente estudiada y sobre la que existe abundante bibliografía. Sí se presenta implementaciones en C++ que hasta ahora no se encontraban públicamente.

4.3.3. Interfaz de usuario

La interfaz de usuario se ha desarrollado en C++, utilizando las librerías Qt (de Trolltech, ahora Nokia). La interfaz presenta información básica sobre el estado del quadrotor, obtenida con un intervalo de tiempo de 500 milisegundos (para no saturar el refresco de información).

Presenta también la vista de una posible cámara adjuntada al quadrotor.

De nuevo el código se encuentra ampliamente comentado, por lo que no es interesante incidir más en esta memoria.



5. Conclusiones

El simulador de quadrotor aquí presentado, es el primer desarrollo en software libre que se realiza para la simulación de este tipo de aeronaves en el mundo de la robótica. Existen desarrollos comerciales, de propósito más general, pero ninguno de libre distribución.

La elección de Gazebo como base para la simulación ha supuesto un acierto, al proponer una plataforma que integra de una manera adecuada y sencilla tres elementos principales en toda simulación: Representación tridimensional, integración de la física de cuerpos rígidos y configuración sencilla para permitir simulaciones versátiles.

El desarrollo, en cuanto a líneas de código ha sido, en cualquier caso, importante. Más de 2.000 líneas de código escritas para proporcionar la funcionalidad adecuada en el controlador, interfaz de usuario y modificaciones del core de Gazebo para permitir las simulaciones de vuelos.

La posibilidad de simular a la misma vez varios quadrotors, junto con otro tipo de objetos o robots (terrestres por ejemplo), lo convierten en una herramienta ideal para el desarrollo y prueba de algoritmos cooperativos, y sistemas multirrobot. La flexibilidad de los mundos permite definir asimismo múltiples entornos, de exterior e interior (outdoor e indoor).

Aún así quedan trabajos futuros: Realizar simulaciones tras pruebas reales, que a modo de repetición, permitan simular con un alto grado de precisión pruebas reales realizadas pudiendo extraer mejores conclusiones, por ejemplo.

6. Referencias

- [1] Design and control of quadrotors with application to autonomous flying. Samir Bouabdallah. Tesis doctoral. École Polytechnique Fédérale de Lausanne. 2007.
- [2] Documentación de Gazebo. <http://playerstage.sourceforge.net/doc/Gazebo-manual-svn-html/>
- [3] Open Dynamics Engine documentación. <http://ode.org/ode-docs.html>
- [4] OGRE Documentación. <http://www.ogre3d.org/docs/api/html/>
- [5] Wikipedia Robot Software. http://en.wikipedia.org/wiki/Robot_software
- [6] Wikipedia Robotics Simulator. http://en.wikipedia.org/wiki/Robotics_simulator
- [7] Wikipedia Player Project. http://en.wikipedia.org/wiki/Player_Project
- [8] <http://en.wikipedia.org/wiki/Quadrotor>
- [9] A 3D Interface for an Unmanned Aerial Vehicle. B. Cervin, C. Mills, and B. C. Wünsche. Auckland, Private Bag 9, University of Auckland, New Zealand.
- [10] Station-Keeping of Quadrotor MAV Using Vision-based Measurement in Hover. Ly Dat Minh, Cheolkeun Ha. Department of Aerospace Engineering. University of Ulsan. Korea. 2009.
- [11] Kinematic visual servo control of a quadrotor aerial vehicle. Odile Bourquardez, Robert Mahony, Nicolas Guenard, François Chaumette, Tarek Hamel, Laurent Eck. Centre National de la Recherche Scientifique. Institut National de Recherche en Informatique et en Automatique – unité de recherche de Rennes. Université de Rennes. 2007.
- [12] A Generic Simulator for Quad-Rotor Unmanned Aerial Vehicles. Robert Barclay. Final Report Undergraduate Dissertation. 2005.
- [12] A Generic Simulator for Quad-Rotor Unmanned Aerial Vehicles. Robert Barclay. Final Report Undergraduate Dissertation. 2005.
- [13] Design and Control of an Indoor Micro Quadrotor. Samir Bouabdallah, Pierpaolo Murrieri, Roland Siegwart.
- [14] PID vs LQ Control Techniques Applied to an Indoor Micro Quadrotor. Samir Bouabdallah, André Noth and Roland Siegwart. Autonomous Systems Laboratory, Swiss Federal Institute of Technology. Lausanne, Switzerland