

**Universidad  
Rey Juan Carlos**

**Grado en Ingeniería en Tecnologías de la  
Telecomunicación**

Escuela Técnica Superior de Ingeniería de Telecomunicación

Curso académico 2016-2017

**Trabajo Fin de Grado**

**APORTES AL ENTORNO DOCENTE DE  
ROBÓTICA JDEROBOT-ACADEMY**

**Autor:** Álvaro Villamil Vuelta

**Tutor:** José María Cañas Plaza

Madrid 2017



# Agradecimientos

En primer lugar, por supuesto a mi familia, sobre todo a mis padres, ya que sin ellos no podría estar aquí.

También a todas las personas que me han acompañado a lo largo de mi vida académica, en el colegio, en el instituto y en la Universidad, sobre todo a los que han pasado de ser simples compañeros a ser amigos, y con los que espero no perder nunca el contacto.

A todos los integrantes del grupo de robótica, en especial a Francisco y Aitor, por su amabilidad y paciencia tanto en los primeros compases como en los problemas de última hora.

Cómo no, a mi tutor José María Cañas Plaza y su infinita paciencia y su constante ánimo en cada pequeño paso que daba, sin su ayuda habría sido imposible acabar este trabajo.

Por último pero no menos importante, a mi pareja, Clara, ya que es la que me ayudaba a desconectar y tener días o ratos de disfrute a pesar de lo duros que han sido los últimos meses, brindándome siempre el mejor ánimo y apoyo que una persona te puede ofrecer.

Y a todos aquellos que hayan contribuido en lo más mínimo y no haya nombrado, gracias de corazón.



# Resumen

Con el avance de la robótica cada vez se hace más necesaria una formación básica en este ámbito. A través del entorno JdeRobot-Academy y sus prácticas se pretende acercar este mundo tan extenso de manera sencilla y asequible a los estudiantes. En este proyecto hemos tratado de ampliar las posibilidades que ofrece este entorno docente ampliando y mejorando el catálogo de prácticas que se presenta al alumno.

Para ello nos hemos servido de diversas herramientas para crear nuevos mundos para robots en el simulador Gazebo, más realistas y más atractivos. Pretendemos mejorar la experiencia de las prácticas con un Fórmula 1 al reproducir de manera realista un circuito carismático y reconocible a primera vista como es el de Mónaco. Para lograrlo reproducimos su trazado y su topología, consiguiendo incorporar a la colección de prácticas mundos en 3D con elevaciones.

Con el mismo objetivo buscamos la forma de añadir a la colección un brazo robótico funcional que pueda ser teleoperado. Nos servimos de brazos ya creados para desarrollar un teleoperador propio que pueda servir en futuras prácticas. Para ello estudiamos ROS, alguna de sus herramientas más avanzadas y la infraestructura del campeonato de brazos robotizados ARIAC.



# Índice general

<b>1. Introducción</b>	<b>1</b>
1.1. Robótica . . . . .	1
1.1.1. Historia de la robótica . . . . .	2
1.2. Educación en robótica . . . . .	6
1.3. Entorno docente JdeRobot-Academy . . . . .	9
<b>2. Objetivos</b>	<b>15</b>
2.1. Descripción del problema . . . . .	15
2.1.1. Requisitos . . . . .	16
2.2. Metodología . . . . .	16
2.3. Planificación temporal . . . . .	18
<b>3. Infraestructura Usada</b>	<b>21</b>
3.1. Blender . . . . .	21
3.2. Simulador Gazebo . . . . .	22
3.3. JdeRobot . . . . .	22
3.4. ROS . . . . .	23
3.4.1. MoveIt! . . . . .	26
3.4.2. rviz . . . . .	26
3.4.3. rqt . . . . .	26
3.5. ARIAC . . . . .	27
3.6. Qt . . . . .	28

<b>4. Circuito de carreras de Fórmula 1</b>	<b>29</b>
4.1. Prácticas con circuito en JdeRobot-Academy . . . . .	29
4.2. Manejo de Blender . . . . .	30
4.2.1. Interfaz . . . . .	31
4.3. Creación de un circuito plano . . . . .	35
4.4. Creación del circuito con elevaciones . . . . .	40
4.5. Mundos para Gazebo . . . . .	44
<b>5. Brazo robótico</b>	<b>53</b>
5.1. Brazos robóticos existentes en el entorno ROS . . . . .	53
5.2. Hablando ROS con un brazo robótico ARIAC . . . . .	56
5.2.1. Nodos y topics . . . . .	56
5.2.2. Nodos, <i>topics</i> y mensajes de un brazo robótico ARIAC . . . . .	57
5.3. Teleoperador de un brazo robotizado . . . . .	64
5.4. Manejo del brazo a través del planificador MoveIt . . . . .	73
<b>6. Conclusiones</b>	<b>79</b>
6.1. Aportaciones . . . . .	79
6.2. Trabajos futuros . . . . .	80
<b>Bibliografía</b>	<b>81</b>

# Índice de figuras

1.1.	Robot de Unimation usado por Ford en 1961. . . . .	3
1.2.	Los primeros robots espaciales, el Mars 3 y el Viking I. . . . .	4
1.3.	Esquema básico del funcionamiento de un robot. . . . .	5
1.4.	Robot ArDrone en Gazebo persiguiendo a otro ArDrone dentro de una práctica. . . . .	11
1.5.	Robot kobuki en Gazebo. . . . .	12
1.6.	Robot de Fórmula 1 en un circuito en Gazebo. . . . .	12
1.7.	Robot taxi en una ciudad en Gazebo. . . . .	13
1.8.	Robot Pioneer en Gazebo frente al conjunto de objetos a escanear. . . . .	13
2.1.	Modelo de ciclo de vida en espiral. . . . .	17
3.1.	Esquema del funcionamiento de JdeRobot. . . . .	24
3.2.	Escenario de la competición ARIAC. . . . .	27
4.1.	Esquema de los componentes de las prácticas con Fórmula 1. . . . .	30
4.2.	Vistas de la práctica de navegación local del F1. . . . .	31
4.3.	Interfaz de Blender. . . . .	32
4.4.	Detalle de Manipuladores de transformaciones en 3D. . . . .	32
4.5.	Detalle del cursor de Blender. . . . .	32
4.6.	Ventana de propiedades. . . . .	35
4.7.	Trazado del circuito de Mónaco usado como plantilla. . . . .	36
4.8.	Detalle de curva Bezier. . . . .	37

4.9.	Trazado del circuito de Mónaco superpuesto a la plantilla. . . . .	38
4.10.	Segmento del circuito. . . . .	39
4.11.	Circuito (sólo el trazado). . . . .	39
4.12.	Circuito plano. . . . .	40
4.13.	Diferentes vistas del circuito de F1 de Mónaco plano. . . . .	41
4.14.	Fondo para el circuito con elevaciones. . . . .	42
4.15.	Circuito con elevaciones. . . . .	42
4.16.	Diferentes vistas del circuito de F1 de Mónaco con elevaciones. . . . .	43
4.17.	Circuitos con una línea roja. . . . .	44
4.18.	Desplegable para exportar ficheros desde Blender. . . . .	45
4.19.	Opciones de exportación de Collada. . . . .	46
4.20.	Coches recorriendo el circuito de Mónaco. . . . .	50
5.1.	Robot PR2. . . . .	54
5.2.	Robot kuka. . . . .	54
5.3.	Robot ur10. . . . .	55
5.4.	Esquema de los componentes de ARIAC centrándonos en el brazo. . . . .	56
5.5.	Grafo de los <i>nodes</i> y <i>topics</i> de ARIAC. . . . .	58
5.6.	Grafo de los <i>nodes</i> y <i>topics</i> de ARIAC. . . . .	58
5.7.	Detalle de las articulaciones del brazo. . . . .	61
5.8.	Interfaz gráfica del controlador del brazo y brazo en la posición inicial. . .	64
5.9.	Ilustración de los grados de libertad del brazo. . . . .	71
5.10.	Ilustración de los grados de libertad de la muñeca. . . . .	72
5.11.	Herramienta rViz mostrando la trayectoria del brazo. . . . .	74
5.12.	Grafo con la relación de nodos y topic de ARIAC, el brazo, MoveIt y rViz. .	75
5.13.	Grafo “simplificado” con la relación de nodos y topic de ARIAC, el brazo, MoveIt y rViz. . . . .	76
5.14.	Lista de mensajes, servicios y acciones que componen la interfaz de MoveIt. .	77

# Capítulo 1

## Introducción

En este proyecto se construirá material docente para la enseñanza de robótica utilizando diversas tecnologías. Para saber cual es el marco de desarrollo y dónde se engloban los conceptos utilizados vamos a introducir algunos términos que servirán como contexto y que permitirán entender mejor de dónde surge la necesidad de estos escenarios y la utilidad de los mismos.

### 1.1. Robótica

Según la Wikipedia[18], *La robótica es la rama de la ingeniería mecatrónica, de la ingeniería eléctrica, de la ingeniería electrónica, de la ingeniería mecánica, de la ingeniería biomédica y de las ciencias de la computación que se ocupa del diseño, construcción, operación, disposición estructural, manufactura y aplicación de los robots.*

Otra definición menos técnica podría ser: La robótica es una ciencia o rama de la tecnología, que estudia el diseño y construcción de máquinas capaces de desempeñar tareas realizadas por el ser humano o que requieren del uso de inteligencia. Por tanto, estamos ante una ciencia que se encarga de diseñar máquinas que sean capaces de reemplazar al ser humano en algunas acciones. Es una disciplina con sus propios problemas, sus fundamentos y sus leyes, y podemos observar dos vertientes de la misma, la teórica y la práctica. En la parte teórica podemos agrupar todas las aportaciones de la informática, la inteligencia artificial y la automatización. En cuanto a la parte práctica observamos aportaciones relativas tanto a la construcción del robot como a la de su gestión, siendo por tanto destacadas las aportaciones de la mecánica, electrónica, programación, etc. Esto hace de la robótica una ciencia con un marcado carácter interdisciplinario.

### 1.1.1. Historia de la robótica

La historia de la robótica ha estado unida a la construcción de “artefactos”, que trataban de materializar el deseo humano de crear seres semejantes a nosotros que nos descargasesen del trabajo. Desde los primeros pasos de la civilización el ser humano ha desarrollado ingenios tanto para facilitar sus labores como para imitar a la naturaleza, fascinando a sus congéneres. De los antiguos egipcios se conservan descripciones de más de 100 máquinas y autómatas, (incluyendo un artefacto con fuego, un órgano de viento, una máquina operada mediante una moneda, una máquina de vapor) en la obra *Pneumática y Autómata* de Herón de Alejandría. Los griegos nos dejaron creaciones como un pájaro de madera, a vapor, que fue capaz de volar, y genios como Leonardo da Vinci el diseño de un *Caballero mecánico*.

En el año 1921 un escritor checo, Karel Čapek, publica su obra “*Los Robots Universales de Rossum*”, en la que aparece por primera vez la palabra “robot” derivada de la palabra checa robota, que significa servidumbre o trabajo forzado. Unos años más tarde, en 1942, la revista americana *Astounding Science Fiction* publica “Círculo Vicioso” (Runaround en inglés), una historia de ciencia ficción escrita por Isaac Asimov donde aparecen por primera vez las Tres leyes de la robótica. Estas leyes establecen lo siguiente:

- 1.<sup>a</sup> Un robot no hará daño a un ser humano o, por inacción, permitir que un ser humano sufra daño.
- 2.<sup>a</sup> Un robot debe hacer o realizar las órdenes dadas por los seres humanos, excepto si estas órdenes entrasen en conflicto con la 1<sup>a</sup> Ley.
- 3.<sup>a</sup> Un robot debe proteger su propia existencia en la medida en que esta protección no entre en conflicto con la 1<sup>a</sup> o la 2<sup>a</sup> Ley.

Pese a que son fruto de una obra de ciencia ficción, estas leyes han dado la vuelta al mundo y multitud de científicos e investigadores las toman como ciertas, siendo un concepto que aún hoy tiene sentido para los futuros desarrollos en torno a sistemas autónomos. La autonomía de las máquinas debería acompañarse de medidas de seguridad que evitan el daño a las personas. Esto es un precepto que las tres leyes de la robótica de Asimov contienen. De hecho, la idea del escritor era proteger al ser humano, que los robots, por muy avanzados que estuvieran, no pudieran volverse contra las personas. En el caso de un coche autónomo, si este conduce sin pasajeros dentro y va a chocar contra otro donde viajan varias personas, ¿debe el primer vehículo echarse a un lado aunque esté circulando correctamente y vaya a sufrir más daños si lo hace? La primera ley de Asimov diría que sí. En cuanto a la segunda ley, actualmente no se concibe el desarrollo de ningún sistema autónomo sin

mecanismos que permitan a las personas manejarlos manualmente, y se considera que aún no existe un sistema tan fiable y preciso como para darle mayor autoridad que al ser humano que lo controla. En el año 1982 Isaac Asimov publicó *El robot completo* (The Complete Robot en inglés), una colección de cuentos de ciencia ficción escritos entre 1940 y 1976. En esta colección vuelve a explicar las tres leyes de la robótica con más ahínco y complejidad moral. Incluso llega a plantear la muerte de un ser humano por la mano de un robot con las tres leyes programadas, por lo que decide incluir una cuarta ley "La ley 0 (cero)": *Un robot no hará daño a la Humanidad o, por inacción, permitir que la Humanidad sufra daño.*



Figura 1.1: Robot de Unimation usado por Ford en 1961.

En la década de 1950 se comienzan a desarrollar los primeros robots comerciales. En 1956 se comercializa el primer robot de la compañía Unimation, fundada por George Devol y Joseph Engelberger. En 1961 se instala en una fábrica de la Ford Motors Company uno de estos robots (*Figura 1.1*), cuya función era la de levantar y apilar grandes piezas de metal caliente. En 1971 el "Standford Arm", un pequeño brazo robótico de accionamiento eléctrico, se desarrolló en la

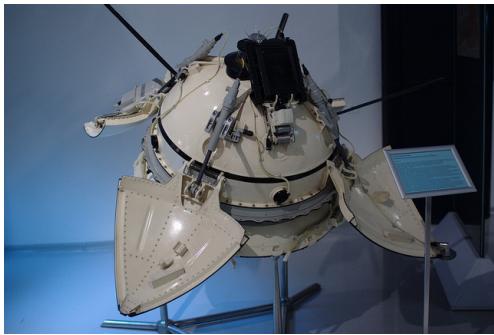
Standford University. En 1973 la empresa Kuka<sup>1</sup> desarrolla el primer robot industrial con seis ejes electromecánicos, el Famulus.

Pero la robótica no se basa sólo en las máquinas que revolucionaron los procesos industriales, el concepto de robótica incluye y cada vez se orienta más hacia los sistemas móviles autónomos, que son aquellos capaces de desenvolverse por sí mismos en entornos desconocidos sin necesidad de supervisión. Con esta idea en mente, en los setenta, la NASA inicio un programa de cooperación con el Jet Propulsion Laboratory para desarrollar plataformas capaces de explorar terrenos hostiles. El primer fruto de esta alianza sería el Mars-Rover, que estaba equipado con un brazo mecánico tipo Standford, un dispositivo telemétrico láser, cámaras estéreo y sensores de proximidad. Sin embargo, fue la Unión Soviética en 1971 la primera en lograr aterrizar un robot en la superficie de marte con éxito, el Mars 3, aunque la comunicación se perdió minutos después. No fue hasta 1976 que la NASA hizo llegar al primer robot estadounidense a Marte, el Viking I. En la Figura 1.2 podemos observar tanto el Mars-Rover como el Viking.

Desde entonces la robótica ha experimentado en multitud de aplicaciones y formatos con modelos sumamente ambiciosos, como es el caso del IT, diseñado para expresar emociones, el COG, también conocido como el robot de cuatro sentidos, el famoso Sojourner o el Lunar Rover, vehículos con control remoto, y otros mucho mas específicos como el Cypher,

---

<sup>1</sup><https://www.kuka.com/>



(a) Mars 3.



(b) Viking I.

*Figura 1.2: Imágenes de los primeros robots espaciales: una copia del Mars 3 (a) expuesto en el Museo Memorial de la Cosmonáutica en Moscú, y en Dr. Carl Sagan posando junto a un modelo del Viking I (b) en el Valle de la Muerte, California.*

un helicóptero robot de uso militar, el guardia de tráfico japonés Anzen Taro o los robots mascotas de Sony. En el campo de los robots antropomorfos (androïdes) se debe mencionar el P3 de Honda que mide 1.60m, pesa 130 Kg y es capaz de subir y bajar escaleras, abrir puertas, pulsar interruptores y empujar vehículos, así como el robot ASIMO de la misma compañía, capaz de desplazarse de forma bípeda e interactuar con las personas.

En general la historia de la robótica la podemos clasificar en cinco generaciones (división hecha por Michael Cancel, director del Centro de Aplicaciones Robóticas de Science Application Inc. en 1984):

1.<sup>a</sup> Generación Robots manipuladores. Son sistemas mecánicos multifuncionales con un sencillo sistema de control, bien manual, de secuencia fija o de secuencia variable.

2.<sup>a</sup> Generación Robots de aprendizaje. Repiten una secuencia de movimientos que ha sido ejecutada previamente por un operador humano. El modo de hacerlo es a través de un dispositivo mecánico. El operador realiza los movimientos requeridos mientras el robot le sigue y los memoriza.

3.<sup>a</sup> Generación Robots con control sensorizado. El controlador es una computadora que ejecuta las órdenes de un programa y las envía al manipulador para que realice los movimientos necesarios.

4.<sup>a</sup> Generación Robots inteligentes. Son similares a los anteriores, pero además poseen sensores que envían información a la computadora de control

sobre el estado del proceso. Esto permite una toma inteligente de decisiones y el control del proceso en tiempo real.

Las dos primeras, ya fueron alcanzadas en los ochenta. La tercera generación, que incluye visión artificial, ha avanzado mucho en los ochenta y noventa. La cuarta contempla movilidad avanzada en exteriores e interiores. Y podríamos hablar incluso de una quinta, en la cual entrarían los más modernos sistemas de aprendizaje autónomo y la inteligencia artificial.

El campo de la robótica está en constante expansión, y su popularidad aumenta rápidamente. Ya no solo vemos grandes avances en los robots industriales, como en cadenas de producción, envasado de alimentos o gestión de almacenes, sino que los robots domésticos están cobrando cada vez más importancia. El éxito de los robots aspiradora como Roomba de iRobot<sup>2</sup>, la incorporación de aparcamiento automático en coches de todas las gamas o incluso asistentes de conducción autónoma como el autopiloto de Tesla<sup>3</sup> o prototipos de Google o Apple, los diversos modelos de robots desactivadores de explosivos de los cuerpos de seguridad del mundo, el sistema quirúrgico Da Vinci que permite incluso operar siendo teleoperado desde otro país, los grupos de robots coordinados usados en construcción o misiones de búsqueda y rescate, termostatos inteligentes como el Nest<sup>4</sup> de Google, o la inmensa variedad de drones del mercado ponen de manifiesto que ésta es una tendencia en auge a nivel global.

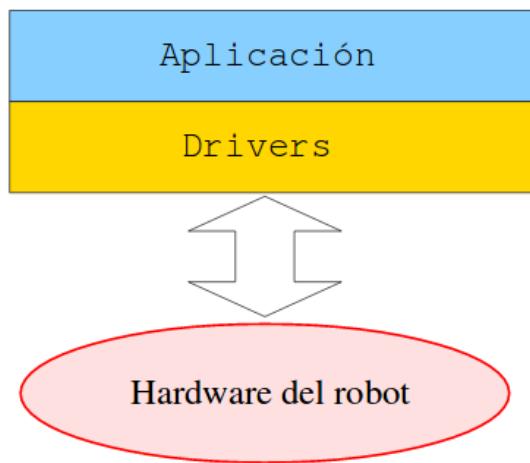


Figura 1.3: Esquema básico del funcionamiento de un robot.

<sup>2</sup><http://www.irobot.es/>

<sup>3</sup>[https://www.tesla.com/es\\_ES/](https://www.tesla.com/es_ES/)

<sup>4</sup><https://nest.com/thermostat/meet-nest-thermostat/>

Viendo el esquema común a cualquier robot (*Figura 1.3*), uno de los factores que permiten diseñar, construir y comercializar los robots asegurando una inteligencia y robustez ante situaciones reales es la programación que hay detrás de estos, su software. Usualmente este software tiene varias capas (drivers, middleware y aplicaciones) y presenta unos requisitos específicos dependiendo de las funciones del robot, de su hardware. En los últimos años se han logrado añadir en la fabricación de estos robots ordenadores personales o microprocesadores, principalmente de bajo coste, y sistemas operativos generalistas, lo que permite aumentar la complejidad de sus tareas y el uso de herramientas estándar de desarrollo. Es también el área de la robótica con mayores expectativas de crecimiento e innovación, y donde más importante es la formación para lograr objetivos cada vez más complejos.

## 1.2. Educación en robótica

El sector de la robótica es un mercado al alza, que demanda científicos e ingenieros de robótica y visión artificial, pero dado que es un campo transversal los profesionales de este sector deben poseer sólidos conocimientos de programación, procesado de imágenes, cálculo, álgebra lineal, métodos numéricos, electrónica y electricidad, etc. Esto hace que se puedan realizar aproximaciones desde diferentes puntos de vista. Uno de ellos, más tradicional, es desde las ingenierías eléctricas y electrónicas. La enseñanza desde estas áreas se centra en la construcción del robot, sus partes móviles y mecánicas, sus sensores, motores, su diseño electrónico, procesadores, etc. Otro acercamiento se realiza desde la Informática, poniendo más énfasis en la programación, dado que la inteligencia del robot una vez construido reside en su software.

Cuando los estudiantes de cualquier nivel educativo construyen un robot están inmersos en un mundo multidisciplinar, dónde la geometría, la trigonometría, la electrónica, la programación, el control, la mecánica, etc, proporcionan las capacidades básicas que redundarán en el éxito de dicha tarea integradora. Aparte de los conceptos que le son propios, la robótica genera entornos propicios para la colaboración, y el trabajo en equipo donde los niños y jóvenes tienen la oportunidad de aprender y practicar las habilidades denominadas como las 4C del siglo XXI<sup>5</sup>:

- Pensamiento Crítico: habilidad imprescindible para un buen aprendizaje. Se basa en la razón efectiva, utilizando varios tipos de razonamiento y analizando la interacción

---

<sup>5</sup>Según la Asociación para las habilidades del siglo XXI (*Partnership for 21st Century Skills*) <http://www.p21.org/>

de las partes de un todo, en el análisis y evaluación de las pruebas, argumentos y puntos de vista, en la interpretación de la información y extracción de conclusiones, y en la resolución de problemas.

- Comunicación: se basa principalmente en la articulación de pensamientos e ideas en diferentes vías, en la escucha eficaz y en el uso de múltiples medios y tecnologías.
- Colaboración: Se basa en la capacidad para trabajar de manera eficaz y respetuosa en diversos equipos, en la voluntad de compromiso en la consecución de objetivos comunes y en la asunción de responsabilidades, tanto compartidas como individuales.
- Creatividad: se basa en el pensamiento creativo, usando técnicas de generación de ideas y elaborando, perfeccionando, analizando y evaluando ideas originales, Y en el trabajo creativo, desarrollando y comunicando nuevas ideas de manera efectiva, siendo abierto y receptivo a nuevas ideas y perspectivas o contribuyendo con ideas creativas en el campo de trabajo.

Actualmente en nuestro país, la robótica aparece en los cursos de secundaria y de formación profesional, aunque se realiza fundamentalmente en la universidad con algunos títulos de grado y postgrado específicos.

La formación en robótica, en términos generales, sólo suele aparecer con suficiente contenido en los programas de formación profesional en las asignaturas técnicas de las ramas correspondientes. En concreto, aparece en el título de Grado Medio de Técnico en Mecanizado (lenguajes de programación de robots), y en una parte del módulo de Sistemas de Control Secuencial en ciclo formativo de Grado Superior del título de Técnico Superior en Sistemas de Regulación y Control Automáticos.

En la enseñanza secundaria la robótica permite acercar la tecnología a los niños y motivarles para aprender conceptos básicos de ciencias, tecnología, ingeniería y matemáticas. Estas áreas han visto reducido el número de estudiantes en los últimos años, y numerosos gobiernos han realizado grandes inversiones para incentivar a los estudiantes a orientar sus estudios hacia estos campos. En la Comunidad de Madrid se ha introducido<sup>6</sup> la asignatura *Tecnología, programación y robótica* en el currículum oficial de la ESO (Educación Secundaria Obligatoria).

Fuera de esta asignatura específica aparece, generalmente asociado a contenidos de automatismos, tan solo en un par de temas de algunas de las pocas asignaturas de tecnología: Tecnología Industrial 3, obligatoria en 3º, y Tecnología Industrial 4, optativa

---

<sup>6</sup>Decreto 48/2015

de 4º curso. En bachillerato, en la modalidad Tecnología, los contenidos de robótica suelen aparecer marginalmente en las asignaturas de Tecnología 1 y 2, del Bachillerato de la Modalidad Tecnología. Generalizando, se puede decir que la mayoría del alumnado no adquiere una formación específica importante en robótica tras su paso por la Enseñanza Secundaria.

En esta etapa se utilizan plataformas como los diferentes robots de LEGO<sup>7</sup> (Mindstorms, NXT, Evo, WeDo, RCX) o placas con procesadores Arduino<sup>8</sup> a las que se conectan diversos sensores, actuadores, servos, etc. Con estas plataformas se introducen las bases de la programación con lenguajes sencillos, usando entornos de prácticas donde completar código ya estructurado o entornos gráficos para niños como Scratch, RCX-code, mbot Blockly. Cabe mencionar el curso de JdeRobot-Kids<sup>9</sup>, que utiliza Arduino como placa programable, mBot como robot móvil y Python como lenguaje para introducir conceptos básicos de tecnología a alumnos jóvenes e iniciarles en robótica y programación, haciéndolo de manera atractiva y enseñando conceptos interesantes de mecánica, electrónica e informática. El carácter del curso es práctico, de *aprender haciendo*.

En la enseñanza superior o universitaria, tradicionalmente se imparten asignaturas de robótica en las escuelas de ingeniería, ya sea industrial, electrónica, informática, etc. La enseñanza de la robótica y materias afines como la visión por computador, la inteligencia artificial y el aprendizaje automático, de interés para nuestra comunidad, encuentra cobijo en las actuales enseñanzas de Grado, principalmente en el ámbito industrial, aunque también en el ámbito informático.

En España existen grados y varios másteres con contenidos en robótica. Una de las titulaciones de Grado en el ámbito industrial que presentan estos contenidos es el Grado en Ingeniería Electrónica y Automática de la Universidad de Zaragoza, que oferta al estudiante, dentro de la tecnología específica de Electrónica Industrial, algunas asignaturas que como Ingeniería de Control, Robótica Industrial o Automatización Industrial, todas relacionadas con la robótica. Adicionalmente esta titulación oferta un módulo optativo de 30 ECTS denominado “Automatización y Robótica”. La titulación de Grado en Ingeniería en Informática de la Universidad de Málaga oferta al estudiante la asignatura “Robótica” de 6 créditos ECTS. En el Grado en Ingeniería Informática de la Universidad de Zaragoza se pueden encontrar las asignaturas “Inteligencia Artificial” y “Aprendizaje”, de 6 créditos cada una.

---

<sup>7</sup>Lego posee toda una gama pensada para la educación: <https://education.lego.com/>

<sup>8</sup><https://www.arduino.cc/>

<sup>9</sup><http://jderobot.org/JdeRobot-kids>

Dentro de los títulos de Máster muchas universidades incluyen más asignaturas de robótica, dado que son estudios más especializados y pueden dedicar una mayor carga lectiva a estos temas. Por ejemplo, el Máster en Ingeniería Informática de la Universidad Carlos III de Madrid presenta entre sus asignaturas “Sistemas de producción automatizados” y “Diseño de Sistemas Inteligentes”, de 3 y 6 créditos ECTS respectivamente. Aunque también existen Másteres totalmente dedicados a este ámbito, como por ejemplo el Máster Universitario en Robótica y Automatización, también de la Universidad Carlos III de Madrid.

Los estudios oficiales de Máster Erasmus Mundus integran un conjunto de cursos de alto nivel que son impartidos por un consorcio formado, por al menos, tres universidades de tres países europeos diferentes. Son cursos integrados porque tienen un plan de estudios conjunto y comportan un período de estudio en al menos dos de las universidades participantes. Este tipo de másteres están pensados para fortalecer la cooperación europea y los vínculos internacionales en la enseñanza superior, y presentan estudios especializados en este sector, como el Erasmus Mundus Masters in VIision and robotics (Universidad de Gerona, Université de Bourgogne y Heriot-Watt University). El Máster VIBOT se estructura en cuatro semestres que proporcionan una formación integral en los ámbitos de la Visión por Computador y la Robótica.

En Estados Unidos, las universidades más punteras en tecnología, como Stanford o el MIT, cuentan con programas de grado y postgrado entre sus planes de estudios.

Las asignaturas de robótica impartidas en la universidad tienen un enfoque práctico, de forma que la interacción del alumno con los robots facilita el aprendizaje y entendimiento de los conceptos teóricos mediante un aprendizaje activo. Es habitual el uso de plataformas específicas para la programación del robot como ROS (sección 3.4) o MATLAB<sup>10</sup> y su paquete Simulink<sup>11</sup>. Otros se centran en el diseño y modelado del robot.

### 1.3. Entorno docente JdeRobot-Academy

Desde la Universidad Rey Juan Carlos se plantea un entorno de enseñanza universitaria llamado *JdeRobot-Academy*, dentro de la plataforma JdeRobot (*Sección 3.3*). Orientado para realizar cursos universitarios de 12-14 semanas, plantea ocultar todo el middleware al alumno y dejar que se centre en la programación de los algoritmos. De esta manera

---

<sup>10</sup><https://es.mathworks.com/products/matlab.html>

<sup>11</sup><https://es.mathworks.com/products/simulink.html>

el alumno desarrolla software para una tarea determinada sin necesidad de desarrollar el software que conecta los elementos del robot, por lo que es perfecto para cursos de introducción a la robótica o que hagan énfasis en la programación de robots ya construidos. En la Universidad Rey Juan Carlos se ha utilizado en la asignatura “Robótica”, en el grado en Ingeniería Telemática, en la asignatura “Visión en Robótica”, en el Máster de Visión Artificial, y en varios cursos de introducción a la robótica y a los drones, así como en el campeonato PROGRAMAROBOT<sup>12</sup>.

Este entorno docente está compuesto de diferentes prácticas independientes que siguen el mismo planteamiento: se propone un problema robótico y el estudiante programa la inteligencia del robot para resolvérla. Se pueden diferenciar una serie de capas que componen dichas prácticas. La capa más baja es donde se encuentra el robot, simulado o real, con todos los sensores o actuadores que lo componen. En la siguiente capa se encuentran los drivers del robot, que permiten acceder a los sensores y actuadores del robot. Y en la última capa se encuentra la aplicación, que analiza los datos de los sensores y da órdenes a los actuadores. En esta capa se encuentra el código de toma de decisiones y planificación. Aquí se encuentra una parte de código específico y necesario para el funcionamiento del robot y otra parte en blanco que el alumno debe completar para resolver con éxito el problema planteado.

Dichas prácticas pueden realizarse sobre robots reales o simulados, aunque generalmente es conveniente comenzar con la simulación antes de pasar a escenarios reales. Se apoyan, por tanto, en el simulador Gazebo (*Sección 3.2*), y usan lenguajes de programación como Python o C++. Aunque el principal sistema operativo para utilizar esta plataforma es Linux, ya sea Ubuntu o Debian, se ha usado la interfaz web de Gazebo para poder lanzarlo cómodamente en Windows y MacOS. Esto se ha conseguido lanzando el servidor de Gazebo y los drivers en un contenedor Docker, el cual permite ejecutar la aplicación de forma nativa en cada sistema operativo y conectarse mediante una aplicación web al simulador Gazebo para ver el mundo de la simulación.

Algunas de las prácticas desarrolladas son:

- *Drones: persecución.* En esta práctica el alumno programa un drone *gato* que persigue a otro drone *ratón* (*Figura 1.4*). El mundo de Gazebo no presenta obstáculos para facilitar la programación del robot, y los drones son similares a los AR.Drone de Parrot<sup>13</sup>. Hay varios niveles de *ratón* disponibles, cuya dificultad de persecución va en

---

<sup>12</sup><http://jderobot.org/Campeonato-programacion-de-robots>

<sup>13</sup><https://www.parrot.com/es/drones/parrot-ardrone-20-elite-edition>

aumento. El drone *gato* posee una cámara frontal, una cámara cenital, inclinómetros y GPS, y facilita una interfaz de control que acepta órdenes simples como avance hacia delante, lateral, ascenso, etc. El objetivo del estudiante es programar los elementos de percepción visual necesarios para que el drone *gato* localice al drone *ratón* y los movimientos necesarios para la persecución del drone *ratón*. También se incluye un componente de evaluación automática y objetiva, el cual puntuá el número de segundos que la distancia con el drone *ratón* está por debajo de un umbral.

Esta práctica sirve para enseñar elementos de control y percepción visual, necesarios para buscar y encontrar al otro drone y para reaccionar en base a sus movimientos., para lo cual son necesarios conocimientos en procesamiento de imágenes y nociones de manejo de OpenCV<sup>14</sup>. También se enseñan nociones como el controlador PID, consistente en un mecanismo de control basado en un algoritmo que, por ejemplo, permite controlar el rumbo del drone frente a cambios de dirección del drone perseguido, y elementos de control basado en casos, por ejemplo, un caso podría ser buscar al robot ya que no aparece en los sensores, y otro modificar la dirección cuando el drone perseguido se acerca a los bordes de la imagen captada por la cámara.

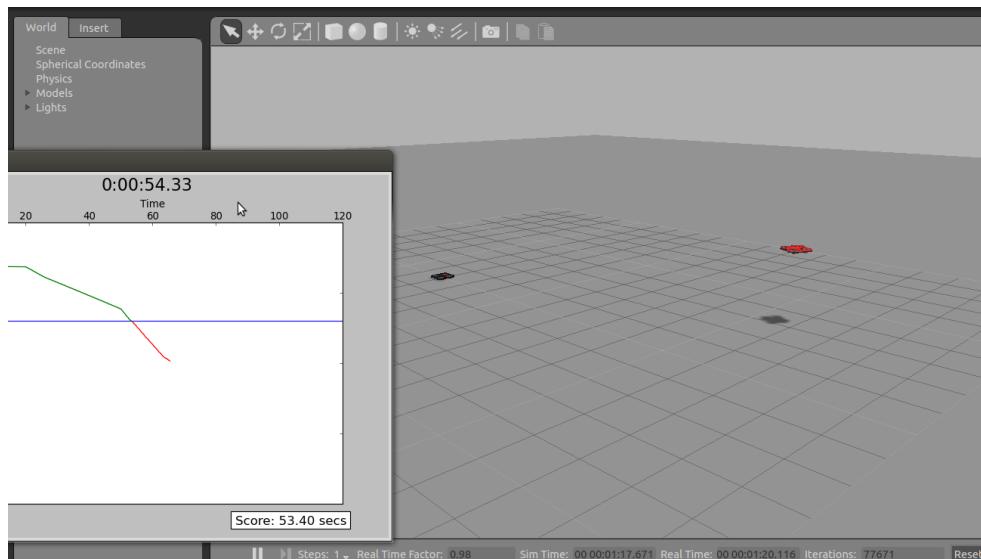


Figura 1.4: Robot ArDrone en Gazebo persiguiendo a otro ArDrone dentro de una práctica.

- *Control visual: sigue líneas.* En esta práctica el alumno debe conseguir que un robot Kobuki (*Figura 1.5*) siga la línea roja de un circuito en el menor tiempo posible. El estudiante debe realizar los filtros de percepción necesarios para que el robot siga la línea roja y los movimientos del robot para mantenerse en la trayectoria adecuada.

<sup>14</sup><http://opencv.org/>

Esta práctica sirve para enseñar técnicas de percepción visual, de control reactivo, control basado en casos y de controladores PID.

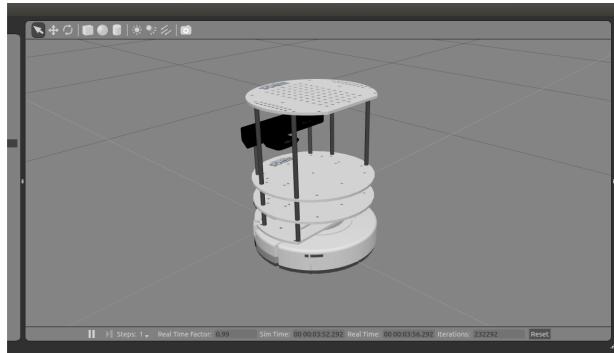


Figura 1.5: Robot kobuki en Gazebo.

- *Fórmula 1: navegación local.* En esta práctica el alumno debe programar un coche de Fórmula 1 (*Figura 1.6*) para que complete una vuelta a un circuito de carreras esquivando los obstáculos que se encuentre en el camino. El robot cuenta con sensores de odometría, GPS y un sensor láser, y la interfaz de movimiento admite órdenes simples como velocidad de avance o de giro.

Para el desarrollo de esta práctica se abordan algoritmos de navegación local como VFF (virtual force field) o campos virtuales, además de técnicas de percepción visual, de control reactivo o controladores PID.

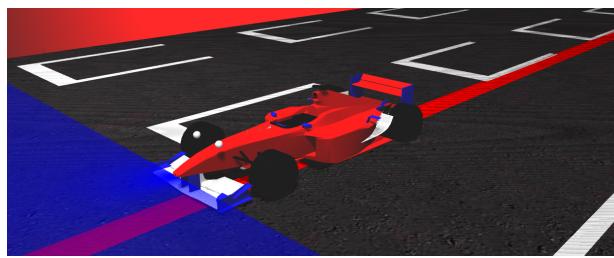


Figura 1.6: Robot de Fórmula 1 en un circuito en Gazebo.

- *TeleTaxi: navegación global.* En esta práctica el alumno debe conseguir que un coche vaya de un punto a otro cualquiera de una ciudad (*Figura 1.7*). El coche, taxi, tiene un sensor GPS y una interfaz como la del Fórmula 1. El alumno debe programar un algoritmo de navegación para que alcance la posición objetivo en un mapa conocido.

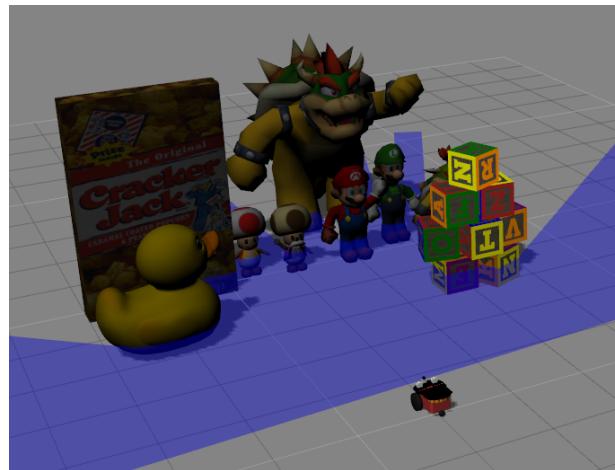
En esta práctica se abordan, además de los elementos de control reactivo, algoritmos de planificación de caminos como Gradient Path Planning. En esta práctica no se necesitan elementos de percepción visual, ya que el taxi se guiará usando el GPS.



*Figura 1.7: Robot taxi en una ciudad en Gazebo.*

- *Visión: reconstrucción 3D.* En esta práctica el alumno debe conseguir que un robot Pioneer (*Figura 1.8*) reproduzca en 3D los elementos que se le presentan. Este robot está equipado con un par estéreo de cámaras. Para conseguir su objetivo, el alumno debe programar un algoritmo de reconstrucción 3D clásico de tres pasos: detección de puntos de interés en las dos imágenes, emparejamiento de píxeles homólogos entre ambas, y triangulación espacial para calcular el punto tridimensional que origina cada pareja de píxeles homólogos.

En esta práctica no se necesita que el robot se mueva, pero sí tiene una alta carga visual, por lo que se abordarán técnicas de procesado de imagen y de percepción visual.



*Figura 1.8: Robot Pioneer en Gazebo frente al conjunto de objetos a escanear.*

Nuestro objetivo con este trabajo es aumentar las posibilidades de esta plataforma, creando nuevos escenarios que aporten valor y funcionalidades nuevas a las prácticas ya

existentes o desarrollando nuevas herramientas que permitan añadir prácticas más variadas a la colección.

En los siguientes capítulos cubriremos todos los elementos necesarios para lograr este objetivo. En el segundo capítulo abordaremos los objetivos marcados para este proyecto, así como los requisitos necesarios para cumplirlos y la metodología que usaremos. En el tercer capítulo expondremos los elementos usados como infraestructura, las herramientas que nos permitirán completar nuestra tarea. En el cuarto y quinto capítulo veremos las soluciones a las que hemos llegado para alcanzar los objetivos planteados, y en el sexto las conclusiones obtenidas de la realización de este proyecto, así como los futuros pasos a seguir.

# Capítulo 2

## Objetivos

Una vez presentado el contexto de este TFG, fijamos ahora los objetivos concretos que se han abordado. También se describe la metodología de trabajo seguida y la planificación temporal.

### 2.1. Descripción del problema

El objetivo principal de este trabajo consiste en mejorar y ampliar la colección de prácticas de JdeRobot-Academy, enriqueciéndolas y aumentando el abanico de posibilidades que se ofrece al alumno. Para ello necesitamos entender el funcionamiento tanto de JdeRobot como de ROS y familiarizarnos con el manejo de programas de edición 3D. Este objetivo lo hemos dividido en dos principales que constituirán nuevas prácticas para el entorno docente:

- Mejorar la infraestructura en Gazebo de las prácticas de JdeRobot-Academy que usan coches de carreras: Crearemos un modelo 3D de un escenario real con elevaciones para realizar las prácticas de JdeRobot-Academy en las que se usa un Fórmula 1, aquella en la que ha de seguir una línea y aquella en la que ha de dar vueltas esquivando obstáculos. Elegimos el circuito de Fórmula 1 de Mónaco por ser un circuito conocido, corto tanto en su longitud como en el área que ocupa, y por estar ubicado en un entorno urbano y montañoso, lo que hace que sea estrecho y con subidas y bajadas. Lo prepararemos para estas dos prácticas específicas, añadiendo barreras en los bordes de la pista o marcas en el asfalto, y lo construiremos de forma que sea sencillo realizar futuras modificaciones para otros tipos de prácticas.

- Diseñar y programar un teleoperador de un brazo robótico en Gazebo: Buscaremos un brazo robótico que funcione en Gazebo y bajo ROS. Analizaremos su construcción y su funcionamiento y construiremos un controlador de bajo nivel que actúe directamente sobre las partes móviles del brazo, es decir, que mueva las articulaciones una a una. Para ello estudiaremos la interfaz de manejo que ofrece el entorno ROS para estos robots articulados. Crearemos una ventana con unos controles intuitivos para facilitar el manejo del robot.

### 2.1.1. Requisitos

Además de alcanzar los objetivos marcados, las soluciones alcanzadas deberán cumplir estos requisitos:

- El desarrollo de este proyecto funcionará para la versión 5.5 del entorno JdeRobot, así como bajo Gazebo 7 y ROS Kinetic. Los componentes software se programarán utilizando principalmente Python, y en aquellos casos que no sea posible, utilizando el lenguaje principal de la plataforma o del tipo de archivo necesario.
- El coste computacional de los mundos 3D creados para Gazebo deberá estar acotado, en la medida de lo posible, para que sea usable desde ordenadores normales, habituales entre los alumnos o en los laboratorios.
- Los mundos creados se incorporarán al repositorio oficial de JdeRobot, por lo que debe ser posible utilizarlos junto con la plataforma de forma eficiente y estar afinados para su uso por terceros.

## 2.2. Metodología

En el desarrollo del software de los componentes de nuestro trabajo utilizamos un modelo de ciclo de vida en espiral. Es un modelo de proceso de software evolutivo, desarrollado por primera vez por Barry Boehm en 1988. En este modelo las actividades a realizar se conforman siguiendo una espiral, en la que cada bucle o iteración representa un conjunto de actividades que completan un modelo del proyecto final. Proporciona un modo de desarrollo evolutivo con un coste y complejidad incremental.

Usando este modelo, al final de cada iteración obtenemos un prototipo funcional que cumple los requisitos marcados para esa iteración, lo que nos permite desarrollar poco a

poco los objetivos. De esta manera aseguramos funcionalidades en cada paso y llevamos un seguimiento más preciso de fallos y bugs. También nos permite adaptarnos mejor a cambios de planificación y de requisitos.

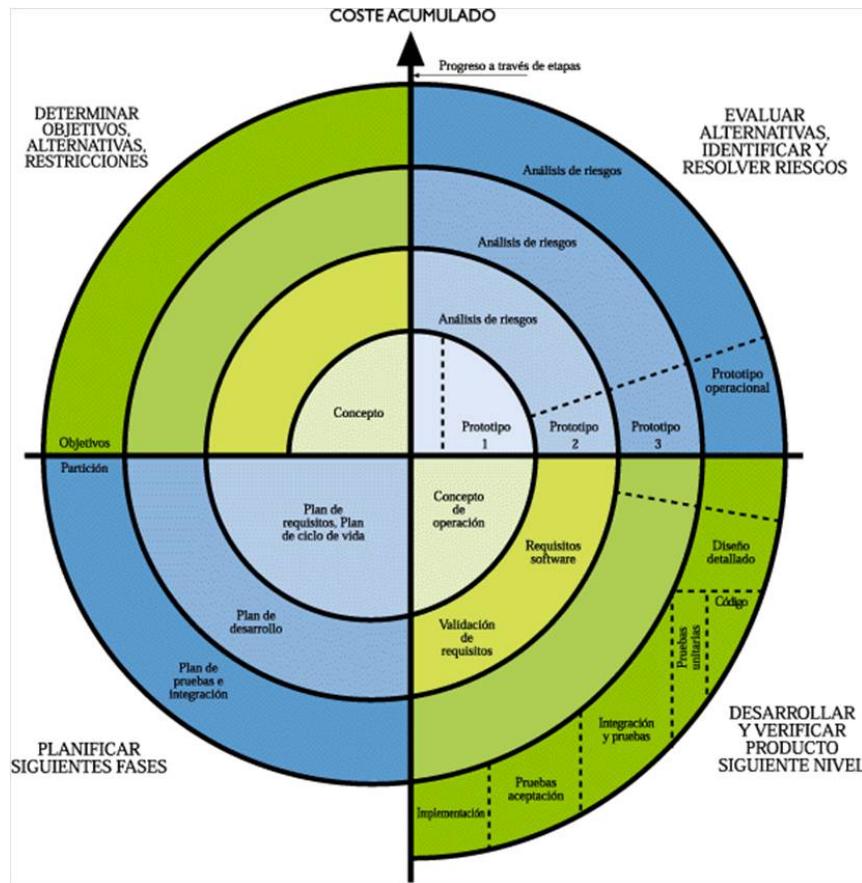


Figura 2.1: Modelo de ciclo de vida en espiral.

En cada vuelta de la espiral se parte de los resultados obtenidos en la anterior y se aumenta la complejidad. Dentro de cada vuelta se siguen una serie de pasos o tareas que guían el desarrollo del proyecto en cada iteración, como muestra el esquema que podemos ver en la figura 2.1. Los pasos son:

1. Determinar objetivos, alternativas, restricciones: En este paso se definen los objetivos específicos que tendremos que cumplir en cada iteración teniendo en mente el resultado final del proceso. Además se diseña una planificación detallada de gestión y se identifican las posibles alternativas.
2. Evaluar alternativas, identificar y resolver riesgos: Se efectúa un análisis detallado de las posibles alternativas que nos permitan cumplir los objetivos marcados utilizando distintos puntos de vista. Además se tienen en cuenta los riesgos del proceso, se elige

la mejor estrategia teniendo en cuenta las dificultades que puedan surgir y se plantean estrategias alternativas para paliar el impacto de estos problemas.

3. Desarrollar y verificar producto: Se desarrolla el producto de acuerdo a la planificación efectuada apuntando a los objetivos propuestos dentro del ciclo. Seguidamente probamos que el resultado sea satisfactorio de acuerdo a dichos objetivos.
4. Planear las siguientes fases: Se revisa el proyecto y se decide si continuar con un nuevo ciclo de la espiral. En caso afirmativo se desarrollan los planes para la siguiente iteración y se subsanan los posibles errores cometidos en la última iteración.

Se han realizado reuniones semanales con el tutor para mantener un seguimiento y una progresión adecuada en el desarrollo del proyecto, alcanzando los objetivos marcados y estudiando diferentes alternativas para conseguirlos.

Para tener constancia de los progresos y del desarrollo general del proyecto se ha utilizado un apartado en la propia página de JdeRobot, un MediaWiki[4] donde ir documentando periódicamente los progresos, incluyendo fotos y vídeos de los logros conseguidos y de los problemas encontrados. Ha servido como referencia para que las reuniones con el tutor estén sincronizadas, y a modo de diario para mantener un registro de los progresos y de las dificultades encontradas. También se ha utilizado un repositorio en GitHub[5] donde se encuentran todos los archivos relacionados con el proyecto, desde test y tutoriales hasta el código fuente del resultado final, pasando por alternativas descartadas y pruebas fallidas.

### 2.3. Planificación temporal

Las fases de desarrollo seguidas en el proyecto se corresponden con los pasos seguidos en los siguientes capítulos. Las principales son:

- Estudio del entorno JdeRobot: En esta primera fase nos centraremos en entender el funcionamiento del entorno JdeRobot y de integrarnos en el grupo de robótica. Estudiaremos los diversos componentes que forman el entorno y qué papel desempeña cada uno, centrándonos en los mundos de Gazebo y los plugins de control de los robots ya que son los elementos que más en profundidad desarrollaremos en nuestro proyecto. Para ello recurrimos a los múltiples ejemplos facilitados y a la lista de correo del grupo de robótica, la cuál resultó ser de una ayuda inestimable para solucionar los problemas surgidos en estos primeros compases del proyecto.

- Desarrollo del circuito realista de Fórmula 1: En esta segunda fase nos familiarizamos con el manejo de Blender, una herramienta potente pero muy diferente a los editores habituales de texto o de imágenes. Recurrimos a diversos tutoriales encontrados en su página centrandonos en el modelado de objetos 3D, ya que características como la animación, la simulación de fluidos o el desarrollo de videojuegos no son necesarias en este proyecto. Nos familiarizamos con el tratamiento de imagen para las texturas, buscando formas de conseguirlas y de editarlas para aplicarlas al circuito. También fue necesaria una investigación sobre las características topológicas del circuito para darle mayor realismo, tanto en la configuración del trazado como en las elevaciones del terreno.
- Estudio de ROS y ARIAC: Al igual que con el entorno de JdeRobot, necesitamos estudiar el funcionamiento de ROS para comprenderlo y aplicarlo en la realización del teleoperador. Encontramos en su extensa documentación y tutoriales toda la información que necesitamos. También estudiamos los componentes que forman ARIAC para comprender su funcionamiento y cómo usar ROS para manejar el brazo. A través de sus tutoriales vemos que ofrece la posibilidad de utilizar planificadores de movimiento como MoveIt, que forman parte de los complementos de ROS, por lo que indagamos más en ellos.
- Desarrollo del teleoperador: Cuando ya hemos adquirido los conocimientos sobre ROS y ARIAC necesarios comenzamos el desarrollo del teleoperador. Para ello repasamos el lenguaje Python y comenzamos dando pequeños pasos para afianzar los conocimientos sobre ROS y relacionarlos con el mundo de ARIAC y el brazo que contiene. Al finalizar realizamos pruebas para comprobar que el funcionamiento es el adecuado y se comporta como esperábamos.
- Documentación del proyecto: Para finalizar, una vez alcanzados los objetivos planteados comenzamos con la escritura de la memoria. Nos apoyamos en el MediaWiki que hemos escrito a medida que realizábamos progresos en el trabajo para recordar los pasos seguidos y las fuentes consultadas.



# Capítulo 3

## Infraestructura Usada

En este capítulo vamos a describir con más detalle todo el software utilizado en la realización del proyecto: programas, plataformas, simuladores, etc. El sistema operativo elegido para desarrollar todo el proyecto ha sido Ubuntu 16.04. Ubuntu es un sistema operativo basado en Debian GNU/Linux y que se distribuye como software libre, el cual incluye su propio entorno de escritorio denominado Unity. El nombre de la distribución proviene del concepto zulú y xhosa de ubuntu, que significa “humanidad hacia otros” o “yo soy porque nosotros somos”. Debido a las similitudes entre los ideales de los proyectos GNU, Debian y en general el movimiento del software libre, y el movimiento sudafricano encabezado por el obispo Desmond Tutu *Premio Nobel de la Paz en 1984* llamado Ubuntu, la compañía británica Canonical Ltd. decidió nombrarlo en honor a dicho movimiento. El eslogan de Ubuntu – “Linux para seres humanos” (en inglés “Linux for Human Beings”) – resume una de sus metas principales: hacer de Linux un sistema operativo más accesible y fácil de usar.

### 3.1. Blender

Blender[6] es un software libre y gratuito de creación en 3D. Está diseñado para realizar tareas como modelado, iluminación, renderizado, animación y creación de gráficos tridimensionales, edición de vídeo y escultura y pintura digital. En Blender, además, se pueden desarrollar videojuegos ya que posee un motor de juegos interno. Actualmente es compatible con todas las versiones de Windows, Mac OS X, GNU/Linux (Incluyendo Android), Solaris, FreeBSD e IRIX. Su interfaz utiliza OpenGL<sup>1</sup> (*Open Graphics Library*)

---

<sup>1</sup><https://www.opengl.org/>

para proporcionar una experiencia consistente y de calidad.

Blender fue liberado al mundo bajo los términos de la Licencia Pública General de GNU v2 (GPL)<sup>2</sup>, y su desarrollo continúa conducido por un equipo de voluntarios procedentes de diversas partes del mundo y liderados por el creador de Blender, Ton Roosendaal.

Se eligió este programa para realizar la edición 3D de mundos para Gazebo frente a alternativas como 3DSMax, Maya o XSI por diversos motivos: es más ligero que sus competidores; posee más herramientas de escultura 3D; la comunidad es muy activa y hay gran cantidad de información, tutoriales y soluciones disponibles; y es de distribución comercial libre y gratuita.

## 3.2. Simulador Gazebo

Gazebo[7] es un simulador 3D de robots para interiores y exteriores, con un motor de físicas y cinemáticas muy potente. Dispone de un conjunto de plugins que facilita la integración con ROS, lo cual agiliza el desarrollo de código y permite la simulación de algoritmos antes de implementarlos en el robot físico, lo cual se puede lograr sin realizar apenas cambios en el código. Además está mantenido por una comunidad activa y la OSRF<sup>3</sup> (*Open Source Robotics Foundation*), la cual también da soporte a ROS, y fué elegido para realizar el DARPA Robotics Challenge<sup>4</sup> entre 2012 y 2015.

Cabe destacar que Gazebo se compone principalmente de un cliente y un servidor. El servidor es el encargado de realizar los calculos y la generación de los datos de los sensores, y puede ser usado sin necesidad de una interfaz gráfica, por ejemplo en un servidor remoto. El cliente proporciona una interfaz gráfica basada en QT que incluye la visualización de la simulación y una serie de controles de multitud de propiedades. Esta configuración permite lanzar multiples clientes sobre un servidor, consiguiendo multiples interfaces de la misma simulación

## 3.3. JdeRobot

JdeRobot[8] es una suite de desarrollo de software de robótica, domótica y sistemas de visión computerizados cuya última versión, la 5.5, es la usada en este proyecto y permite

---

<sup>2</sup><https://www.gnu.org/licenses/old-licenses/gpl-2.0.html>

<sup>3</sup><http://www.osrfoundation.org/>

<sup>4</sup><http://www.theroboticschallenge.org/>

la integración con ROS Kinetic. Proporciona un entorno distribuido donde las aplicaciones se forman mediante una colección de componentes asíncronos. Estos componentes utilizan interfaces ICE<sup>5</sup> para comunicarse, lo que permite lanzarlos desde distintos equipos y que estén escritos en diferentes lenguajes, como C++, Python o Java.

JdeRobot simplifica el acceso a elementos de hardware, siendo tan simple como realizar una llamada a una función. También permite que los sensores o actuadores con los que se comunica sean reales o simulados, conectados mediante la red tanto dentro de la misma máquina como en una red local o de forma remota mediante internet. Actualmente se han desarrollado drivers para una multitud de dispositivos, como por ejemplo sensores RGB como Kinect o cámaras USB o IP, vehículos como Kobuki o Pioneer, drones como el ArDrone de Parrot, etc.

Como podemos ver en la figura 3.1, los componentes programados dentro de JdeRobot se comunican con los robots mediante interfaces ICE, como el componente del círculo superior de la figura. En el circulo izquierdo podemos ver cómo cada interfaz ICE se comunica con un elemento del robot, como la cámara, los sensores de posición o los motores. En este caso son plugins pertenecientes a un robot simulado en Gazebo, que permite estudiar el comportamiento en entornos controlados. Como vemos en el círculo de la derecha, los mismos interfaces ICE se comunican con el elemento equivalente de este robot, que en este caso es un robot real y los elementos son los drivers de los sensores y actuadores reales. De esta manera, el mismo componente JdeRobot sirve para teleoperar un drone ArDrone Parot simulado en el ordenador y uno real en el exterior.

Es un software libre, licenciado como GPL y LGPL que se sirve de software como Gazebo, ROS, OpenGL, QT... que se usa tanto para docencia como para investigación en la URJC y ha formado parte del *Google Summer of Code 2015*<sup>6</sup>, programa donde Google premia a los estudiantes al completar un proyecto de programación de software libre durante un verano.

## 3.4. ROS

El Sistema Operativo de Robots, ROS[11] (*Robot Operating System*) es un entorno de trabajo flexible donde desarrollar software para robots. Es una colección de herramientas y librerías que buscan simplificar la tarea de crear comportamientos robustos y complejos en una amplia variedad de plataformas robóticas.

---

<sup>5</sup><http://www.zeroc.com/>

<sup>6</sup><https://summerofcode.withgoogle.com/organizations/6493465572540416/>

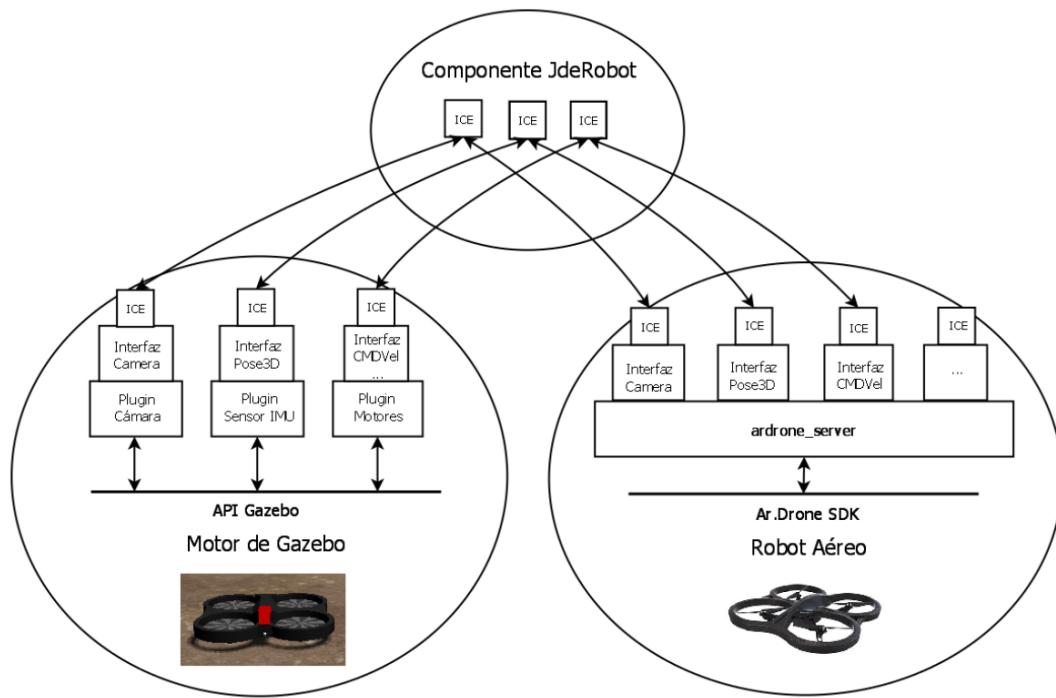


Figura 3.1: Esquema del funcionamiento de JdeRobot.

Desde mediados de la década de los 2000 se han realizado esfuerzos para aunar los entornos y herramientas existentes en sistemas de software dinámicos y flexibles para su uso en robots. Estos esfuerzos culminaron, gracias al interés y la ayuda de innumerables desarrolladores, en las ideas que forman el núcleo de ROS y sus paquetes de software fundamentales. Desde entonces se ha expandido su uso bajo la licencia de software libre BSD<sup>7</sup> y se ha convertido en una plataforma muy usada en la comunidad de investigadores.

Desde el inicio ROS se desarrolló en multitud de instituciones para multitud de robots, permitiendo a una gran variedad de investigaciones tener éxito bajo esta plataforma. Sólo el núcleo de ROS se mantiene en los servidores centrales, cualquier desarrollador es libre de crear, desarrollar y compartir sus propias ideas y proyectos de forma que, si así lo desea, estén disponibles para toda la comunidad desde sus propios repositorios. De esta forma se consigue mantener un ecosistema formado por decenas de miles de usuarios a nivel global, desde proyectos como hobby hasta sistemas industriales automatizados.

ROS se diseñó para ser modular y fragmentado, de modo que los usuarios pueden usar sólo las partes que necesiten. En bajo nivel ofrece una interfaz de comunicación por mensajes que permite ahorrar tiempo manejando los detalles de la comunicación entre nodos mediante un mecanismo anónimo de publicación/suscripción de mensajes

<sup>7</sup>[https://es.wikipedia.org/wiki/Licencia\\_BSD](https://es.wikipedia.org/wiki/Licencia_BSD)

estructurados. Este sistema fuerza al usuario a implementar interfaces limpias entre los nodos del sistema, mejorando la encapsulación y promoviendo la reutilización de código.

Adicionalmente ROS proporciona librerías y herramientas para agilizar el trabajo de sus usuarios. Dado el carácter colaborativo y comunitario del proyecto, se han unificado una gran variedad de formatos mensajes estándar que cubren la mayoría de las necesidades básicas en robótica, tales como posiciones, transformaciones, vectores, sensores como cámaras o lasers, datos de navegación como caminos o mapas, etc.

Un problema común que aborda ROS es la descripción de un robot de forma que sea comprensible para un ordenador, consiguiendo un Formato Unificado de Descripción del Robot o URDF (*Unified Robot Description Format*). Consiste en un fichero XML en que se describen las propiedades físicas del robot, partiendo del cual el robot se puede utilizar con librerías, simuladores y planificadores de movimientos.

También proporciona herramientas de diagnóstico, estimación, localización, navegación, así como una colección de herramientas gráficas y de línea de comandos para facilitar el desarrollo y la depuración. Las herramientas de línea de comandos permiten la utilización de ROS desde cualquier terminal, incluso con conexión remota. Las herramientas gráficas incluyen rviz y rqt, muy potentes tanto para planificar como para desarrollar proyectos en ROS.

Para entender mejor cómo funciona ROS podemos pensar en un sistema de grafos en el que situamos los siguientes elementos:

- Nodos: Son las partes de código que se ejecutan. Escritos en C++ o Python permiten realizar tareas en el robot, suscribirse y publicar en topics o proporcionar y usar servicios. De esta manera se facilita el diseño modular de los proyectos.
- Topics: Son las vías de comunicación usadas por los nodos. Cada topic utiliza un único tipo de mensaje, de esta manera un nodo puede utilizar varios topics para comunicarse.
- Mensajes: Son los datos estructurados que se envían entre topics. En ROS existen varios tipos definidos para los mensajes más utilizados, pero se pueden definir nuevos tipos de mensajes de acuerdo con las necesidades particulares de cada proyecto
- Servicios: A diferencia de los topics, son vías de comunicación síncronas entre nodos compuestas por dos mensajes: uno de petición y otro de respuesta. De esta forma el nodo que envía la petición espera hasta recibir la respuesta.

### 3.4.1. MoveIt!

MoveIt![13] es un software de código abierto para ROS (Robot Operating System) que es el estado de la técnica de software para la manipulación móvil. De hecho, podríamos afirmar que se está convirtiendo en un estándar de facto en el campo de la robótica móvil, ya que hoy en día más de 65 robots utilizan este software.

Incluye diversas utilidades que aceleran el trabajo con brazos robóticos, y sigue la filosofía de ROS de reutilización de código. Este software permite llevar a cabo tareas de planificación de trayectorias complejas, percepción 3D, cálculos cinemáticos, control de colisión, control y navegación de forma sencilla, accediendo por la API o mediante las herramientas de la consola.

### 3.4.2. rviz

Rviz[14] es una herramienta de visualización en 3D llamada que posibilita que prácticamente cualquier plataforma robótica pueda ser representada en imagen 3D, respondiendo en tiempo real a lo que le ocurre en el mundo real. Se puede usar para mostrar lecturas de sensores y obtener información de estado de ROS.

Usado en conjunto con MoveIt! permite mostrar el brazo en su estado actual, la colocación del brazo en una posición objetivo, y la visualización del camino pensado por MoveIt! y del movimiento real del brazo siguiendo dicho camino.

### 3.4.3. rqt

Rqt[15] es un software de ROS que implementa varias herramientas de GUI (*Graphical User Interface*) en forma de plugins, permitiendo cargarlas unificadas como una ventana en la pantalla facilitando trabajo al usuario. Simplemente con un comando en la consola, *rqt* muestra una ventana donde elegir cualquier plugin disponible en el sistema en ese momento.

Contiene una herramienta que ha resultado muy útil para la realización de este proyecto: *rqt\_graph*. Al introducir en la consola *rosrun rqt\_graph rqt\_graph* crea un grafo dinámico que muestra qué nodos y qué topics están activos en ese momento y cuál es su relación. Al situar el ratón encima de cada elemento marcará con un código de color cuál es el elemento activo, de qué tipo es y cuál es su relación con los demás elementos del grafo.

### 3.5. ARIAC

ARIAC[9] (*Agile Robotics for Industrial Automation Competition*) es una competición pionera cuyo objetivo es probar la agilidad de los sistemas robóticos industriales. Realizada por primera vez en Junio de 2017, nace de un esfuerzo conjunto entre la Conferencia de Automatización en Ciencia e Ingeniería del IEEE o CASE (*Conference on Automation Science and Engineering*) y el NIST (*National Institute of Standards and Technology*). Se sirve de Gazebo como plataforma de simulación y de un conjunto propio de modelos, plugins y scripts para simular un brazo robótico en un entorno dinámico, todo ello elaborado con la ayuda de la OSRF (*Open Source Robotics Foundation*). Tiene el objetivo de aumentar la productividad y la autonomía de los robots industriales, entendiendo como agilidad la cosección de manera automática de identificación de fallos y recuperación de los mismos, automatización para disminuir la reprogramación ante cambios en la producción, interacción con el entorno incluso en áreas no previstas inicialmente, y la capacidad de *plug and play*, es decir, de introducir robots de otros fabricantes sin la necesidad de reprogramarlos.



Figura 3.2: Escenario de la competición ARIAC.

El objetivo de dicha competición es resolver una serie de problemas dentro del escenario proporcionado. En la Figura 3.2 podemos observar el escenario donde los participantes realizarán las pruebas. Consta de un brazo manipulador ur10 (*Figura 5.3*) situado sobre un carril que le permite desplazarse, cámaras situadas en las zonas importantes, de una cinta transportadora a un lado del robot y de diversas bandejas al otro. El objetivo de las pruebas es recoger los objetos que aparecen en dicha cinta transportadora y depositarlos en la bandeja correspondiente, según las instrucciones de cada prueba. La competición consta de tres fases clasificatorias y una final que tendrá lugar en el mes de Junio de 2017.

En este proyecto nos servimos del entorno y del propio brazo robótico para desarrollar nuestros propios plugins para entender y controlar el brazo por medio de ROS.

### **3.6. Qt**

Qt[16] es un framework multiplataforma orientado a objetos ampliamente usado para desarrollar programas (software) que utilicen interfaz gráfica de usuario, así como también diferentes tipos de herramientas para la línea de comandos y consolas para servidores que no necesitan una interfaz gráfica de usuario.

Qt es desarrollada como un software libre y de código abierto a través de Qt Project, donde participa tanto la comunidad, como desarrolladores de Nokia, Digia y otras empresas. Qt se distribuye bajo los términos de licencias como GNU Lesser General Public License.

Utiliza el lenguaje de programación C++ de forma nativa, aunque se pueden utilizar otros lenguajes de programación como Python añadiéndolos a través de bindings. Funciona en las principales plataformas y tiene un amplio apoyo. El API de la biblioteca cuenta con métodos para acceder a bases de datos mediante SQL, así como uso de XML, gestión de hilos, soporte de red, una API multiplataforma unificada para la manipulación de archivos y una multitud de otros métodos para el manejo de ficheros, además de estructuras de datos tradicionales.

# Capítulo 4

## Circuito de carreras de Fórmula 1

Una vez explicado el contexto, los objetivos y las herramientas usadas en el proyecto, en este capítulo vamos ver en detalle la construcción de un mundo para Gazebo que se pueda usar en las prácticas de JdeRobot-Academy que emplean coches de carreras, en concreto el circuito de Fórmula 1 de Mónaco.

### 4.1. Prácticas con circuito en JdeRobot-Academy

Las prácticas del entorno JdeRobot-Academy están diseñadas de forma que tanto el mundo como el robot están correctamente montados y configurados, y la aplicación que controla al robot está parcialmente completa. El alumno no necesita realizar modificaciones al mundo simulado o al robot, pero debe completar el código de la aplicación para realizar las tareas propuestas en la práctica. En la figura 4.1 podemos ver la relación entre los componentes software de estas prácticas, así como el espacio destinado al trabajo del alumno. Marcado con una flecha roja esta Gazebo, para señalar las partes modificadas con este Trabajo Fin de Grado. No cambiamos nada del coche ni de la aplicación docente, sí cambiamos el mundo de Gazebo para lograr que la misma práctica se situe en otro escenario.

Actualmente el entorno JdeRobot-Academy presenta dos prácticas basadas en circuitos de Fórmula 1:

- Práctica de navegación local autónoma: En esta práctica el Fórmula 1 dispone de unos sensores que le permiten detectar obstáculos. En la figura 4.2 podemos apreciar el funcionamiento del sensor comparando la información que recibe con la vista real

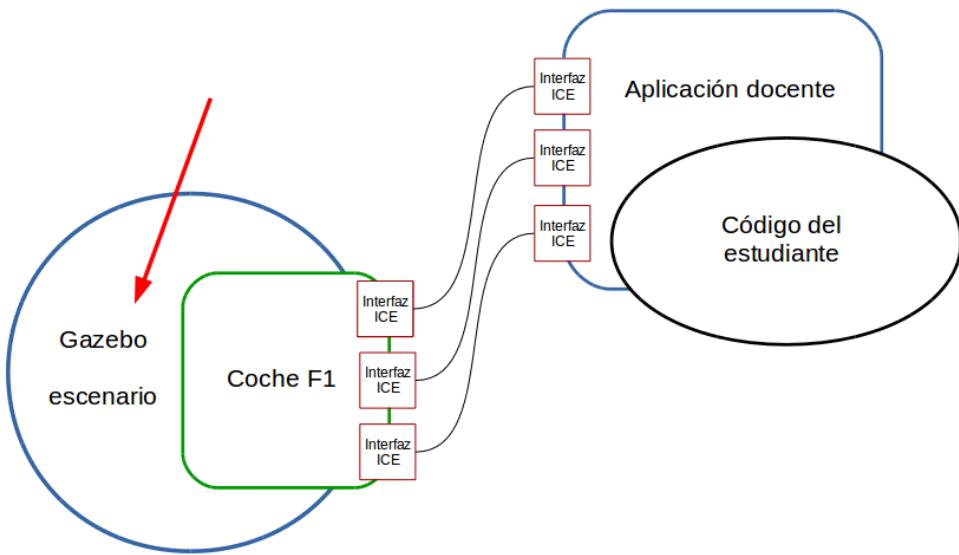


Figura 4.1: Esquema de los componentes de las prácticas con Fórmula 1.

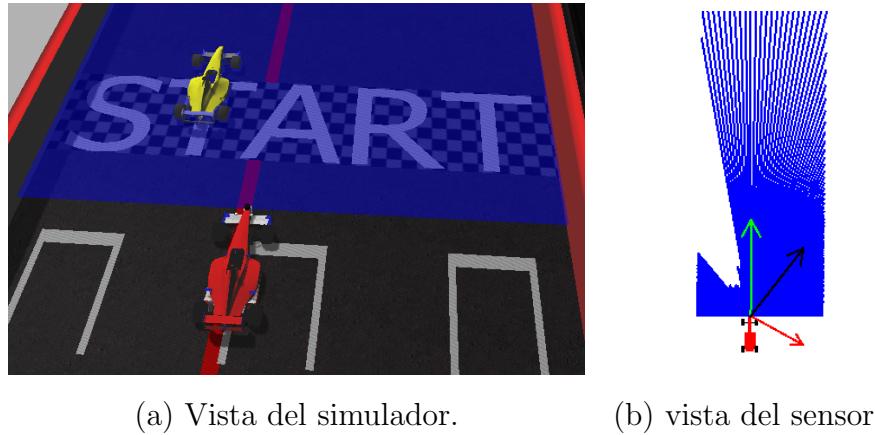
de la simulación. En esta práctica el alumno ha de rellenar el código con elementos de percepción, que le permitan identificar los obstáculos, y con algoritmos de navegación local como el VFF (visual force field), logrando dar vueltas al circuito esquivando los obstáculo presentes.

- Práctica de sigue-línea: En esta práctica el Fórmula 1 dispone de una cámara con la que capta imágenes del circuito. En este caso el circuito dispone de una línea roja en el suelo, y el coche ha de seguir esa línea para completar las vueltas. En esta práctica el alumno ha de programar el código con algoritmos de procesamiento de imagen, como filtros de color o de formas, y con elementos de control como controladores PID para que el robot siga la línea.

Nuestro objetivo es reconstruir el circuito de Mónaco de forma que se pueda utilizar en ambas prácticas, creando para ello mundos de Gazebo que se acoplen al esquema de funcionamiento de estas prácticas, como muestra la flecha de la figura 4.1.

## 4.2. Manejo de Blender

En esta sección vamos a explicar cómo se utiliza este potente programa de creación, renderizado y animación de gráficos tridimensionales. El uso de este tipo de programas es, a priori, de los más difíciles, ya que trabajar sobre un mundo tridimensional en una



*Figura 4.2: Vistas de la realización de la práctica, tanto lo que muestra Gazebo (a) como una representación de lo que recibe el sensor (b).*

pantalla de ordenador, desplazando un ratón sobre una mesa en dos dimensiones, exige un esfuerzo de abstracción considerable.

Es por ello que en los primeros pasos de esta sección nos dedicaremos a presentar la interfaz del programa y exponer la inmensa cantidad de opciones y herramientas de que dispone, centrándonos en aquellas que han resultado relevantes para la consecución de los objetivos marcados.

#### 4.2.1. Interfaz

Como se verá, la interfaz de Blender no sigue el patrón típico de los programas a los que estamos habituados, como editores de texto y hojas de cálculo o entornos de desarrollo de software, por lo que resulta fácil desorientarse al principio.

La interfaz de usuario de Blender está compuesta por 4 ventanas por defecto, como se pueden diferenciar en la figura 4.3. Cada ventana tiene una cabecera con las herramientas adecuadas para trabajar sobre dicha ventana y, a su vez, cada herramienta está dotada de sus correspondientes pestañas para una completa edición. Esta disposición facilita y agiliza el uso apropiado del programa. A su vez, cada cabecera de cada ventana tiene el botón Tipo de Editor, mediante el cual se puede cambiar el tipo de ventana que se muestra, por lo que se puede personalizar el aspecto para agilizar el trabajo.

En primer lugar tenemos la ventana de Vista 3D, bordeada en color Rojo. En esta ventana se visualiza todo el trabajo y los cambios que se realizan con el programa. En esta ventana podemos observar los siguientes objetos y menús:

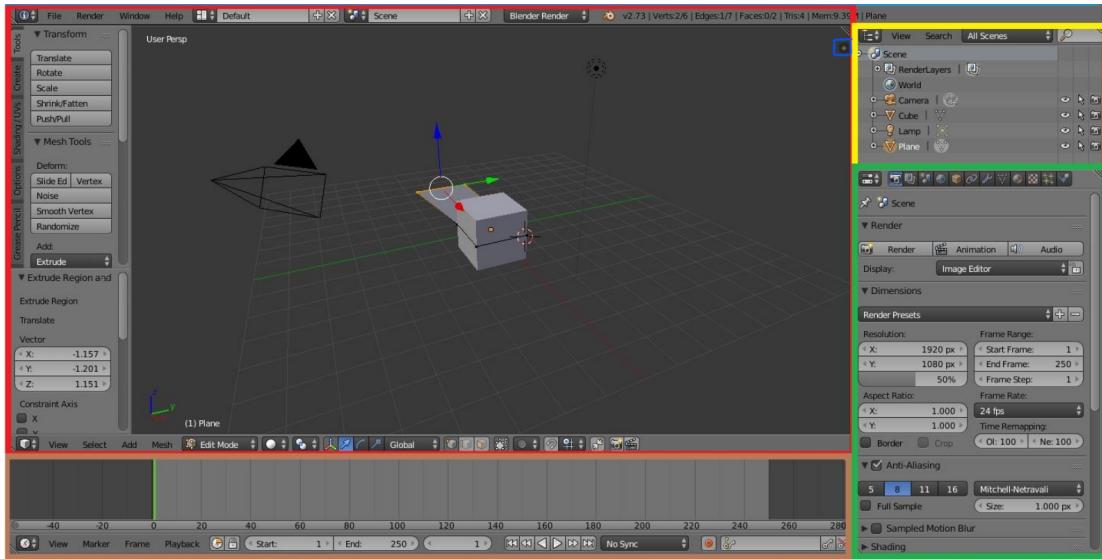


Figura 4.3: Interfaz de Blender.

- Manipuladores de Transformaciones en 3D (*Figura 4.4*): Muestra de manera visual información de las transformaciones a realizar. Cada objeto puede ser transformado de tres maneras: translación (G), rotación (R) y escalado (S). Utilizando la combinación Ctrl+Space, o bien haciendo clic en el ícono del sistema de coordenadas, se puede mostrar y ocultar el manipulador.



Figura 4.4: Detalle de Manipuladores de transformaciones en 3D.

- Cursor 3D (*Figura 4.5*): El cursor 3D es una herramienta muy útil y se utiliza para una variedad de cosas como representar el lugar donde se añadirán nuevos objetos o representar el punto de pivote para una rotación. Sin embargo, es muy engorroso a la hora de trabajar ya que es muy fácil moverlo accidentalmente y al necesitar usarlo se pierde bastante tiempo reubicándolo.

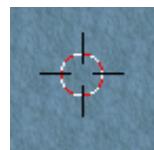


Figura 4.5: Detalle del cursor de Blender.

- Cubo: Al iniciar Blender, por defecto, aparece un cubo situado en el centro de la ventana 3D. Es uno de los elementos básicos que se pueden insertar y la mayoría de

las veces basta este simple cubo para comenzar a trabajar.

- Luz (de tipo lámpara): Al iniciar Blender, por defecto, también aparecerá una Luz de tipo lámpara que estará en algún sitio cerca del centro de la escena. Es posible crear otras fuentes de luz, como un sol en el cenit de la escena o una luz direccional que ilumine todos los objetos. En cualquier caso, aunque en los diferentes modos de edición no sea necesario, a la hora de renderizar la escena para visualizar el resultado del trabajo es necesaria alguna fuente de luz, si no, aparecerá una escena llena de siluetas.
- Cámara: Al iniciar Blender, por defecto, aparecerá una cámara que estará en algún sitio por el centro de la ventana 3D y, probablemente, enfocando al cubo. En la figura 4.3 es esa especie de pirámide de vértices negros a la izquierda del cubo. A la hora de renderizar es necesaria una cámara.
- Objeto seleccionado actualmente: Este campo, situado en la parte inferior izquierda de la escena, al lado del eje de coordenadas, muestra el nombre del objeto seleccionado actualmente.
- Modo Edición: Este botón, en forma de desplegable, se encuentra a la izquierda de los botones de manipuladores de transformaciones 3D. Da acceso a un modo de edición para manipular la geometría del objeto. Al activarse aparecen disponibles tres botones de opciones, justo a la derecha de los botones de manipuladores de transformaciones 3D. Nos permiten cambiar entre la selección y edificación de los vértices, de las aristas y de las caras del objeto. La posibilidad de cambiar entre estas opciones de selección es especialmente útil a la hora de dar forma a los objetos que componen el circuito.
- Vista de la escena: Este botón, en forma de desplegable, se encuentra a la derecha del botón de Modo Edición. Da acceso a un desplegable que permite elegir la vista de la escena 3D entre las posibles, como sólida, texturas, materiales, "wireframe", renderizada, etc. Nos permite ver los objetos de la escena con sólo las texturas, sólo los vértices, ya renderizado, etc, para poder trabajar más cómodo en según que condiciones y ver los resultados de aplicar texturas o del renderizado final.

Para realizar con comodidad el trabajo sobre cualquier tipo de elemento existen diferentes vistas disponibles, cada una con un atajo de teclado propio, en concreto del teclado numérico. De esta forma para el "1" el programa proporciona una vista frontal de la escena; con el número "3" se obtiene una vista derecha; etc. Una de las más cómodas

para realizar el trabajo es la que le corresponde al número “7” ya que se trata de una vista cenital, que una vez superpuesto el plano del circuito facilitan la tarea de trazar el recorrido. Por último, con la tecla “5” podemos cambiar la vista de Perspectiva a Ortográfica. La vista Perspectiva es la más similar a la realidad, dando profundidad a la escena y sensación de lejanía y proximidad de los objetos. La vista Ortográfica es más artificial, pues es como una proyección de la anterior, pero es útil en determinadas situaciones para no alterar las proporciones o manejar con fluidez objetos que quedarían tapados de otra forma.

Esta ventana posee un botón de propiedades propio marcado en color Azul en la parte superior derecha, el cual despliega la ventana de propiedades (*Figura 4.6*). Esta ventana muestra las propiedades del objeto seleccionado y resulta muy útil a la hora de añadir nuevos bloques o realizar modificaciones de los ya existentes, pudiendo modificar la localización, rotación, escala, etc.

Debajo de la ventana de vista 3D, bordeada en marrón, se encuentra por defecto la ventana de Línea Temporal. Aquí se reflejan cronológicamente los bloques u objetos que se han añadido o modificado y es muy utilizada al trabajar con animaciones. En nuestro caso la hemos sustituido por otra que se adapta mejor a nuestras necesidades.

A continuación, bordeada en amarillo, se puede identificar la ventana de Objetos y Jerarquías, donde se pueden ver todos los datos que se utilizan en el trabajo. De esta forma se pueden controlar los diferentes bloques que se utilicen, las luces que se añaden a la escena, cámaras y toda clase de elementos disponibles en la escena. En esta ventana se pueden seleccionar directamente los elementos que se deseen independientemente, y realizar acciones sobre ellos como restringir o habilitar la visualización, selección o renderización de dicho elemento. Esto supone una gran ayuda cuando se superponen diferentes elementos en la ventana.

Debajo de ésta, marcada en color verde, se encuentra la ventana de Propiedades, en la cual se pueden editar las propiedades de los bloques, objetos, materiales, texturas etc. que se utilizan en el trabajo. Esto se consigue mediante los llamados botones de contexto, los cuales muestran un grupo de paneles con opciones diferentes para cada botón. Gracias a esta disposición se esconden multitud de herramientas muy usadas en un espacio reducido y cómodo. Algunas de las tareas que permiten llevar a cabo son: asignar el material o la textura deseada a los elementos creados o modificaciones para replicar, deformar, dividir o desdoblzar elementos entre muchas otras opciones



Figura 4.6: Ventana de propiedades.

### 4.3. Creación de un circuito plano

Una vez presentadas las opciones básicas de Blender y cómo desenvolverse entre ellas con un mínimo de soltura pasaremos a desarrollar el proceso de creación del circuito de Mónaco. En este proyecto comenzamos creando el trazado del circuito sin elevaciones, para coger soltura en el manejo del programa y explorar las diferentes opciones antes de realizar tareas más complejas.

Vamos a replicar en Blender los diferentes tramos y elementos del circuito como distintos objetos con textura, materiales, etc. Los juntaremos con un objeto plano compuesto que también tiene el mar para conseguir recrear el trazado y sus alrededores.

Comenzamos eliminando el cubo que aparece por defecto, desplegando la ventana

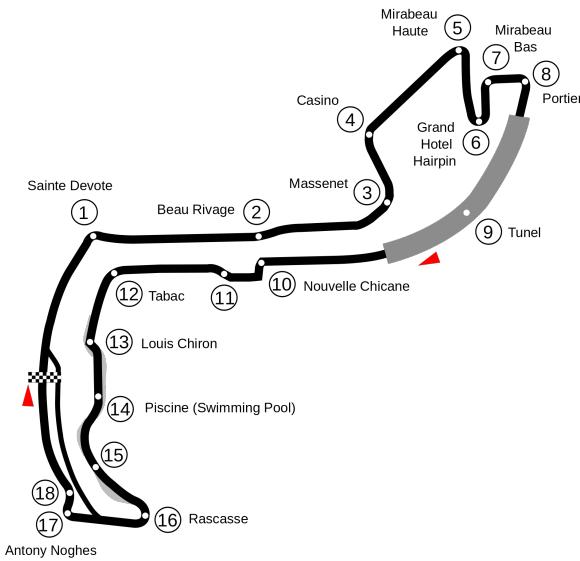


Figura 4.7: Trazado del circuito de Mónaco usado como plantilla.

de propiedades del menú principal y buscando el apartado de “Background Images”. Activamos la casilla y buscamos la imagen que deseamos poner de fondo, en este caso la imagen de la figura 4.7. Es el trazado real del circuito, visto desde arriba. Lo usaremos como plantilla creando curvas y tramos de carretera en Blender que se solapen con estas. Al hacer click en el botón “Add Image” podremos seleccionar la imagen que queremos ver de fondo, así como el eje en el que la queremos ver. Esta herramienta es especialmente útil ya que nos permite ver una superposición de la imagen del trazado al usar la vista cenital, que corresponde con la tecla “7” del teclado numérico, pero en cualquier otro ángulo no se verá, con lo que facilita enormemente la tarea de escalado y modelado del circuito.

A continuación añadimos una curva “Bezier” (*Figura 4.8*), hacemos esto desde el menú superior en *Add, Curve, Bezier*. Esto creará un nuevo objeto, que podemos renombrar en la ventana de Objetos para facilitar el acceso y la selección posteriormente, cuando tengamos mas objetos en la escena. Este tipo de curvas, como se puede apreciar en la figura 4.8, son unas curvas com muchos elementos que nos permiten moldearlas a nuestro gusto. En cada extremo del tramo se componen de una recta con tres puntos. el central que establece el inicio del trazo de la curva, y los otros dos que establecen el giro de la curva y lo pronunciado que és. Cuanto más cerca estén del punto central más cerrado será el ángulo de giro, y cuanto más alejados más suave la curva descrita.

Al hacer click con el botón izquierdo del ratón mientras mantenemos la tecla Control pulsada añadimos un nuevo punto a la curva ya existente. De esta forma podemos ir

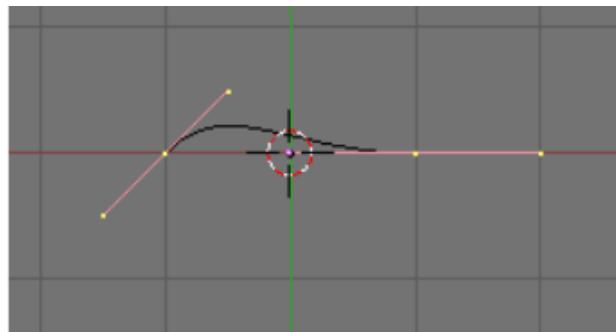


Figura 4.8: Detalle de curva Bezier.

añadiendo puntos y deformando la curva hasta que se superponga con la imagen del trazado. Así logramos una curva cerrada correspondiente al trazado sobre la cual situar los elementos que compondrán la carretera, como podemos apreciar en la Figura 4.9 (*La línea naranja es la curva Bezier que será el trazado*).

A continuación añadimos un plano haciendo *Add, Mesh, Plane*, lo cual generará un cuadrado plano en el centro de la escena. A continuación seleccionamos una arista y la extruimos (*estiramos*) dos veces, una más corta y una más larga. hacemos lo mismo con la arista opuesta del cuadrado. Es importante realizar este paso ayudándonos de las flechas de colores (*verde, rojo y azul*) que aparecen al seleccionar un objeto para mantener el conjunto en el mismo plano. Una vez realizado, extruimos los dos rectángulos más pequeños hacia arriba, consiguiendo crear la carretera, las vallas y una acera para el circuito, aunque de momento sólo aparecen en gris.

Para ayudarnos en la labor de texturizar objetos, editamos el tipo de ventana que es la de la parte inferior, el gráfico temporal, y lo sustituimos por un Editor de Imagen/UV, el cual posee herramientas avanzadas de manejo de imágenes para texturizado de superficies. De esta manera vamos cara por cara de nuestro objeto seleccionándola en la vista 3D, añadiendo una textura<sup>1</sup> en el Editor UV, y observando los resultados en la vista 3D cambiando el modo de visualización a texturizado. La ventana del Editor de imagen nos sirve para redimensionar las texturas, girarlas, pintar encima de ellas, establecer qué zonas aparecen en la superficie seleccionada, y multitud de herramientas que hacen mucho más sencilla la edición de texturas. Una vez finalizado el paso de texturizar las caras del objeto obtenemos el segmento de la figura 4.10.

Es muy importante crear un material para cada textura y asignarlo a las caras correspondientes, de otra manera no se aplicará la textura al renderizar y será como

---

<sup>1</sup>Las imágenes usadas como texturas y materiales en este proyecto han sido obtenidas de manera gratuita de la página textures.com[17]

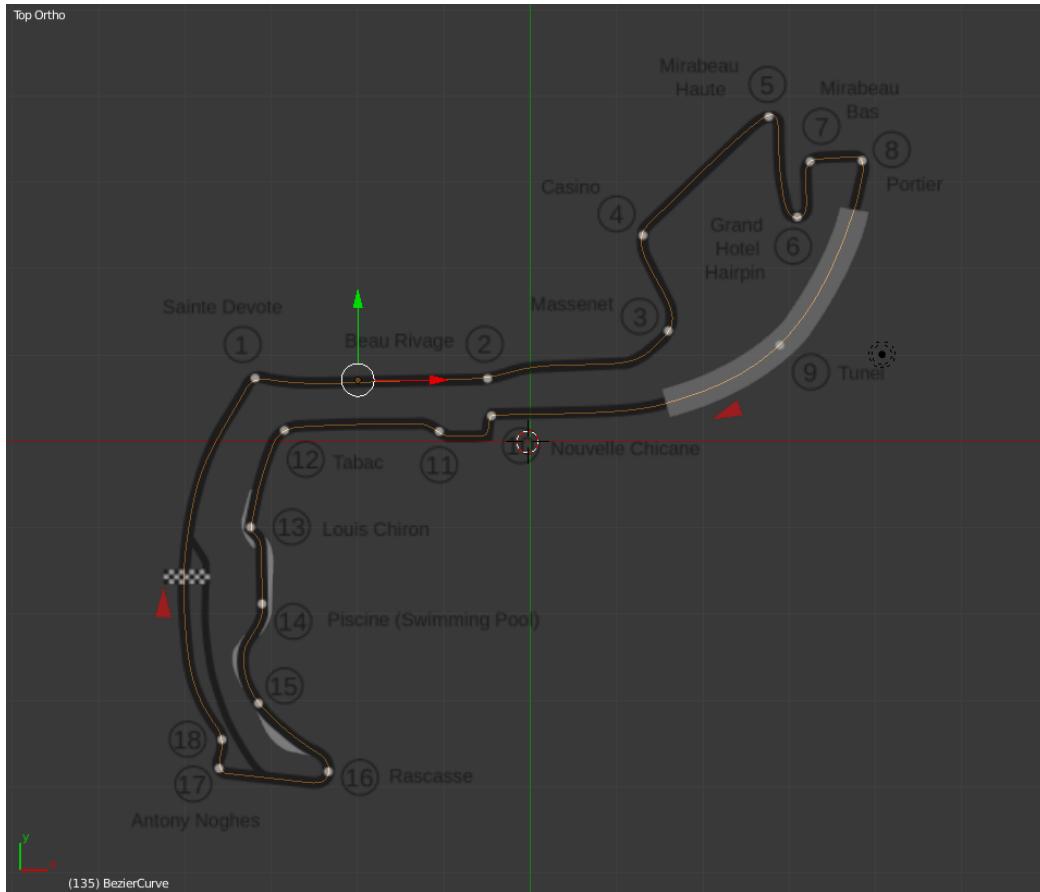
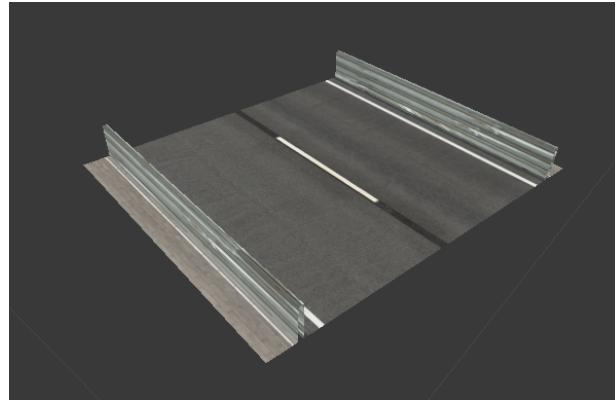


Figura 4.9: Trazado del circuito de Mónaco superpuesto a la plantilla.

haber trabajado en vano. Esto se consigue mediante la ventana de propiedades y haciendo click en la pestaña de materiales. Es necesario crear un material nuevo, añadir como fuente la misma imagen que se ha usado como textura anteriormente, renombrar para no confundirlos más adelante, y asignarlo a la superficie correspondiente. Además, en la pestaña texturas, debemos repetir los pasos para crear la textura de la misma forma que el material. Necesitaremos además asignar cada textura al material correspondiente y cambiar el tipo de mapping a UV. Es una tarea laboriosa pero muy importante, ya que de no realizarse correctamente las texturas podrían no aplicarse y el objeto aparecería gris plano.

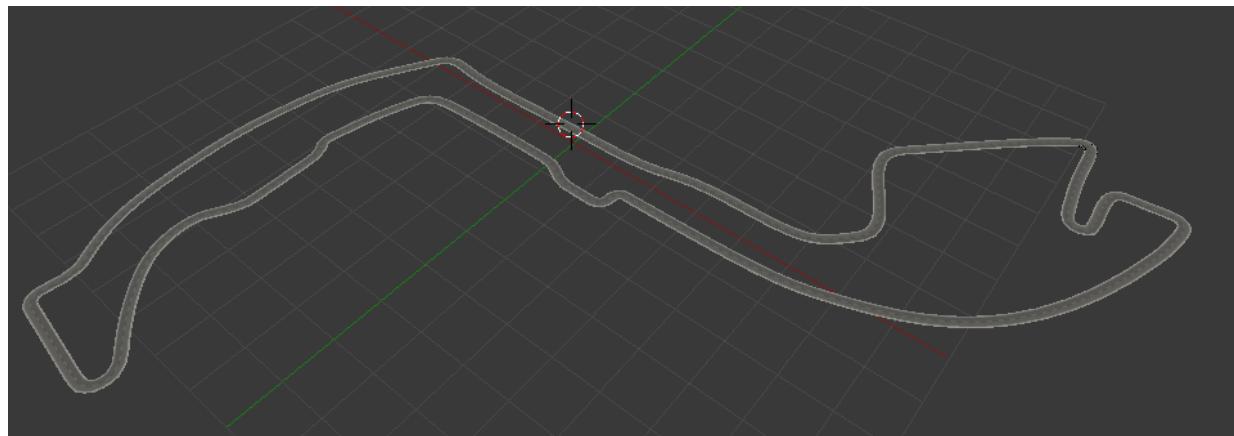
Una vez realizados todos estos pasos seleccionamos el segmento del trazado creado y vamos a la ventana de propiedades y hacemos click en la pestaña de modificadores. Añadimos un nuevo modificador, un Array. Con esto conseguimos crear multitud de objetos iguales conectados entre sí, como si de un circuito de *Scalextric* se tratase. Despues añadimos otro modificador, una curva en este caso, y elegimos como elemento modificante la curva Bezier que hemos creado anteriormente. De esta manera, jugando con el número de copias que realiza el array, conseguimos que el segmento creado se repita y deforme



*Figura 4.10: Segmento del circuito.*

siguiendo el trazado de la curva, adquiriendo la forma del circuito deseado. Este método requiere de muchos retoques manuales, ya que en los vértices de las curvas cerradas el programa no puede calcular bien y solapa y deforma de manera incorrecta los vértices del segmento. Una vez realizados todos los retoques, así como la unión del primer y último segmento, obtenemos el objeto de la figura 4.11.

Una vez obtenido el primer trazado completo necesitamos añadir un plano que servirá de fondo para el circuito. Para ello añadimos un nuevo objeto plano a la escena, lo aumentamos de tamaño hasta que se pueda situar el circuito en su interior y lo subdividimos en cuadrículas. Esto lo hacemos así para poder asignar texturas a cada cuadrícula independientemente y simular de manera más realista que este circuito se sitúa en un puerto, estando a la orilla del mar casi la mitad de su recorrido. Una vez realizada la división deformamos algunos vértices escondiéndolos debajo del trazado, necesitando así menos divisiones y aligerando la carga de procesamiento para la simulación del circuito. Una vez asignadas todas las texturas, repitiendo los pasos descritos para texturizar el



*Figura 4.11: Circuito (sólo el trazado).*

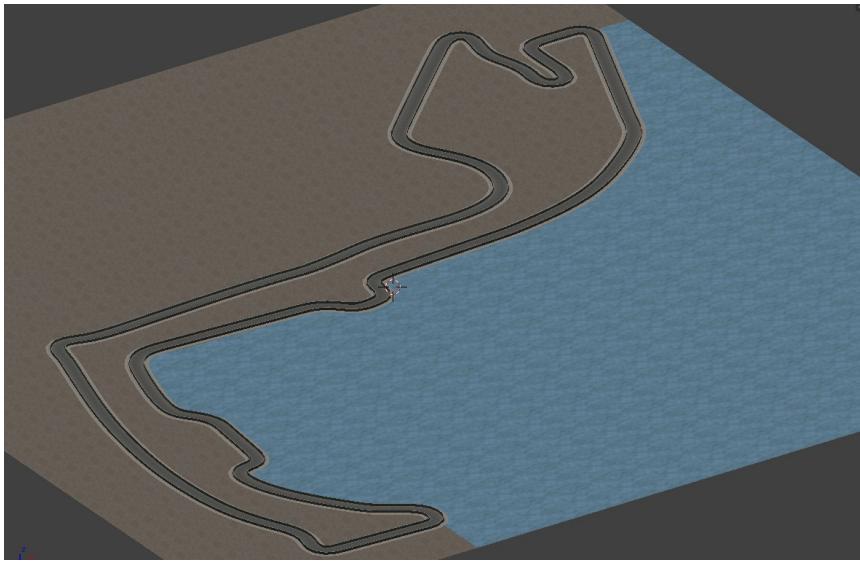


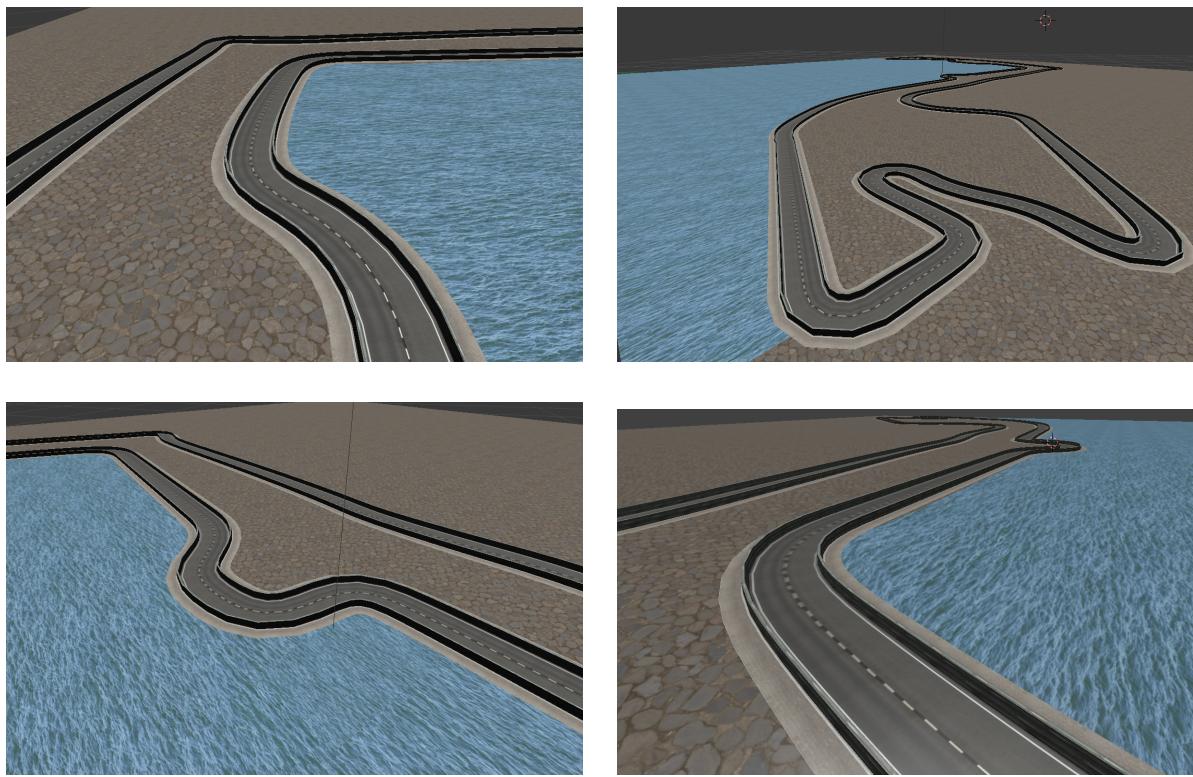
Figura 4.12: Circuito plano.

segmento del circuito, obtenemos el circuito de Mónaco (*Figura 4.12*), el cual podemos ver en detalle en la Figura 4.13.

#### 4.4. Creación del circuito con elevaciones

Una vez modelado el circuito plano pasamos a modelarlo ahora con elevaciones. Dada la dificultad de simplemente añadirlas al circuito ya creado, partimos de cero en la creación de este nuevo escenario. Puesto que ahora es mucho más relevante el fondo, comenzamos por esa parte. Eliminamos el cubo por defecto y añadimos un nuevo objeto, ésta vez una *Mesh*, *Grid*, que llamaremos rejilla. Es diferente del plano por que ya está subdividido, como una rejilla. Al crearlo, en la parte inferior izquierda de la ventana de vista 3D, aparecen una serie de propiedades únicas de este objeto que modificamos a nuestro gusto. En este caso el número de subdivisiones de la rejilla y el tamaño de ésta. Elegimos una cantidad moderada de divisiones ya que, aunque van a afectar visualmente a que las elevaciones no sean todo lo suaves que desearíamos, van a condicionar la carga de procesamiento del mundo en Gazebo, y queremos que sea todo lo ligera posible para poder realizar prácticas con múltiples objetos y vehículos sin que se vea comprometido el rendimiento.

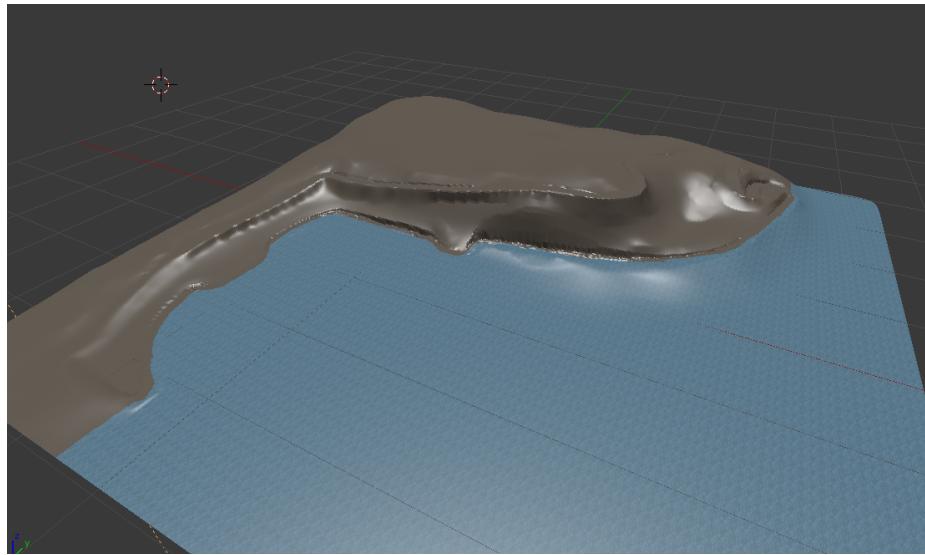
Haciendo click en el botón de Modo Edición accedemos a otro modo, en este caso de Escultura. Una vez en este modo, en la parte superior izquierda de la ventana de vista 3D se despliega una multitud de herramientas y opciones para esculpir los objetos de la escena. Como lo que nos interesa es elevar la rejilla de forma que simule las alturas y elevaciones



*Figura 4.13: Diferentes vistas del circuito de F1 de Mónaco plano.*

del circuito real de Mónaco, accedemos a la opción de bloqueo y seleccionamos los ejes x e y, con lo que sólo modificaremos la altura del plano sobre el que proyectaremos el trazado. Modificando las opciones del pincel hasta dejarlo a nuestro gusto comenzamos a esculpir la rejilla. Usando la misma imagen de antes como plantilla de fondo y fotos reales del circuito esculpimos un prototipo de lo que serán las elevaciones, que más tarde retocaremos para ajustarnos mejor a las peculiaridades del trazado. También nos serviremos de otro tipo de pincel para alisar las rugosidades creadas al modelar de forma más basta y así suavizar el terreno para acomodar mejor al circuito.

A continuación creamos de nuevo, siguiendo los mismos pasos, la curva Bezier que servirá de trazado. Creamos ahora un plano, que pintamos de un color llamativo, y extendemos a lo largo de la curva bezier, consiguiendo un trazado de referencia para acabar de dar los últimos detalles a la rejilla. Tanto a la curva como a este plano le aplicamos como modificador la rejilla, consiguiendo darles altura y pudiendo ver el recorrido real del circuito con las elevaciones modeladas. Ahora modelamos la rejilla teniendo en cuenta el recorrido del circuito y su situación a orillas del mar, por lo que tenemos mucho cuidado de mantener plana toda la parte baja y del mar y aplicamos las elevaciones en las demás partes. Una vez modelado texturizamos de forma análoga a como hicimos anteriormente.



*Figura 4.14: Fondo para el circuito con elevaciones.*

Una vez modelado y texturizado el plano, eliminamos el trazado de referencia y obtenemos el suelo sobre el que descansará el circuito (*Figura 4.14*).

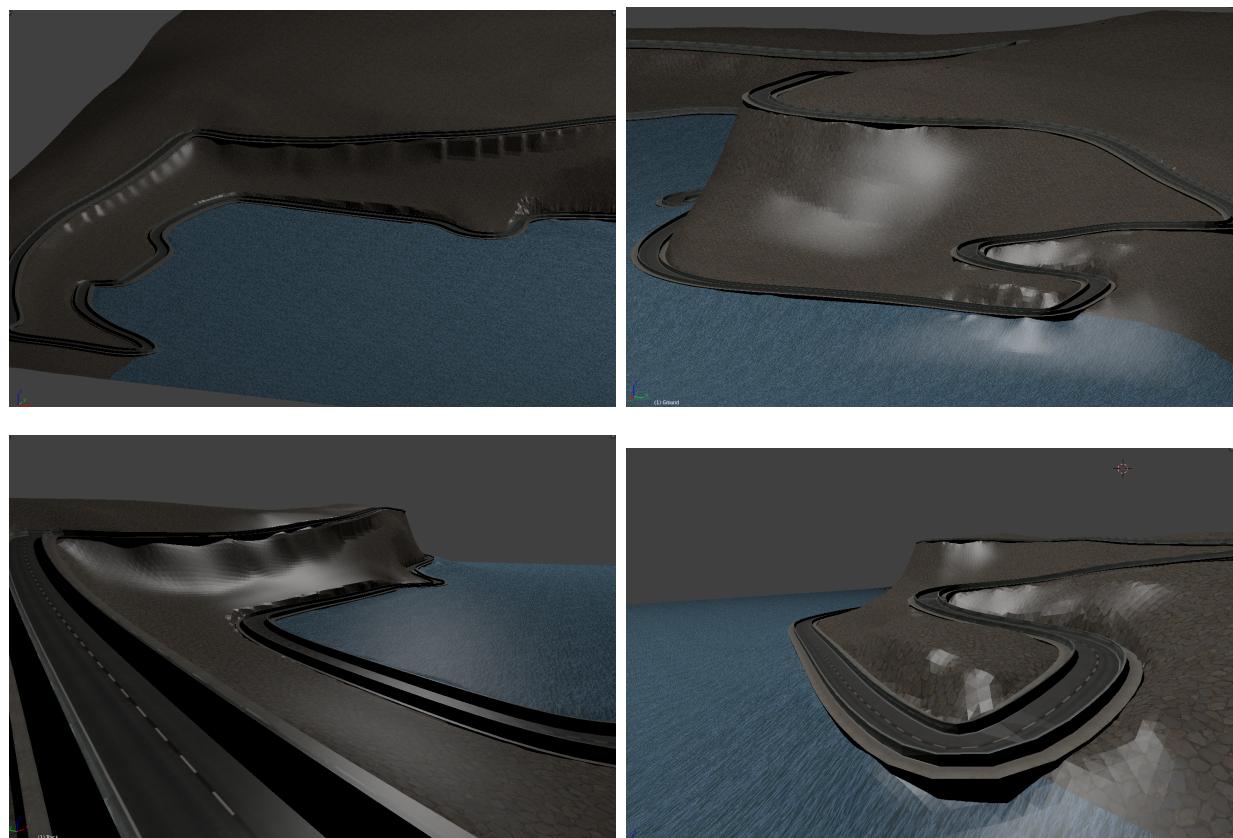
Ahora repetiremos todos los pasos de creación del trazado (creamos el segmento, lo texturizamos, lo extendemos a lo largo de la curva) y además le aplicamos la rejilla como modificador. Si hemos realizado bien el paso anterior con el trazado de referencia, el trazado final debería descansar sobre la caja realizada y acoplarse perfectamente al terreno, de no ser así retocamos nuevamente la rejilla con las herramientas de escultura.



*Figura 4.15: Circuito con elevaciones.*

Una vez completado este paso podemos dar por finalizado el circuito (*Figura 4.15*). En la Figura 4.16 se pueden ver detalles del circuito donde se aprecian mejor las elevaciones del terreno.

Dado el carácter práctico de este escenario, y dado que ya existen prácticas desarrolladas con fines similares, procedemos a crear una variante del circuito, tanto el plano como

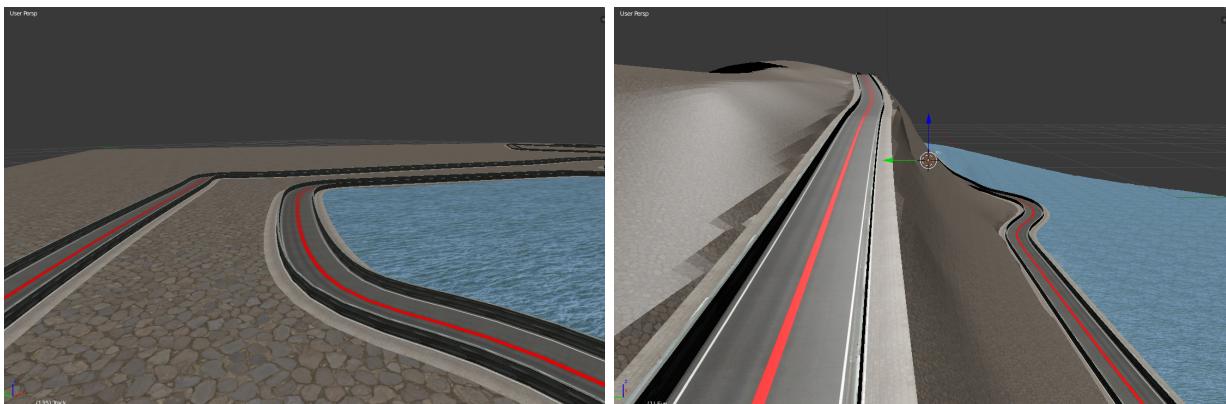


*Figura 4.16: Diferentes vistas del circuito de F1 de Mónaco con elevaciones.*

el elevado, con una línea roja en el centro de su recorrido. Dicha línea servirá para realizar la práctica de control visual en la que es necesaria una referencia clara de un color determinado, siendo el rojo un color claramente diferente de cualquier componente del circuito como el asfalto o las vallas. Como podemos ver en la Figura 4.17 el circuito es idéntico en ambos casos salvo por la linea roja en el medio del asfalto. Para realizar esta modificación de la manera más eficiente y práctica realizamos unos pocos cambios al modelo original. Primero guardamos la escena con un nombre diferente para diferenciarlos. Después editamos la textura usada para crear el material del asfalto con un programa como GIMP<sup>2</sup> y le dibujamos esa línea roja. Después, en el nuevo fichero de Blender cambiamos, tanto en la textura como en el material asociado al asfalto, la imagen de fuente por la editada. Una vez hecho esto en ambos circuitos conseguimos esta variante más orientada a prácticas específicas.

---

<sup>2</sup>Siglas de “GNU Image Manipulation Program”, gratuito y de código abierto. <https://www.gimp.org/>



(a) Mónaco plano.

(b) Mónaco con elevaciones.

Figura 4.17: Circuitos de mónaco editados para contener una línea roja en su trazado, tanto el plano (a) como el que contiene elevaciones (b).

## 4.5. Mundos para Gazebo

Ya hemos conseguido modelar dos escenarios, un circuito de Mónaco plano y otro con elevaciones, más realista. Pero aún no nos sirven, hemos de integrarlos en Gazebo para que puedan ser usados. Para ello estudiamos la estructura de los ficheros \*.world de Gazebo, y vemos que se sirven de modelos en 3D en ficheros \*.sdf y \*.conf para componer los mundos. SDF<sup>3</sup> es un formato XML que describe entornos y objetos para simuladores robóticos, visualización y control. Creado originalmente como parte de Gazebo, fué diseñado pensando en aplicaciones robóticas científicas, y ha evolucionado hasta convertirse en un formato robusto, estable y extensible capaz de describir objetos estáticos y dinámicos, terrenos, luces o físicas. Al analizar la estructura de los ficheros \*.sdf observamos que importan los objetos que componen el mundo de otros ficheros en formatos como .mesh, .model, .dae, etc. Al comparar con otros mundo ya creados dentro de la plataforma JdeRobot vemos que muchos modelos están en formato .dae<sup>4</sup>. Así pues elegimos dicho formato para exportar el objeto desde Blender, comenzando primero con el circuito plano sin línea. Para ello hacemos click en el menú *File, Export, Collada* (Figura 4.18).

En alguna versión de Blender esta opción no está disponible, por lo que hay que añadirla manualmente instalando desde fuente la extensión o buscar otra versión. En nuestro caso inicialmente instalamos la versión 2.76<sub>b</sub> y no tenía disponible esta opción. Optamos por instalar desde fuente la versión 2.78<sub>c</sub> que sí lo tiene.

<sup>3</sup>Página de SFD: <http://sdformat.org/>

<sup>4</sup>Extensión asociada a los archivos Collada, esquema XML para transportar objetos 3D entre aplicaciones. <https://www.khronos.org/collada/>

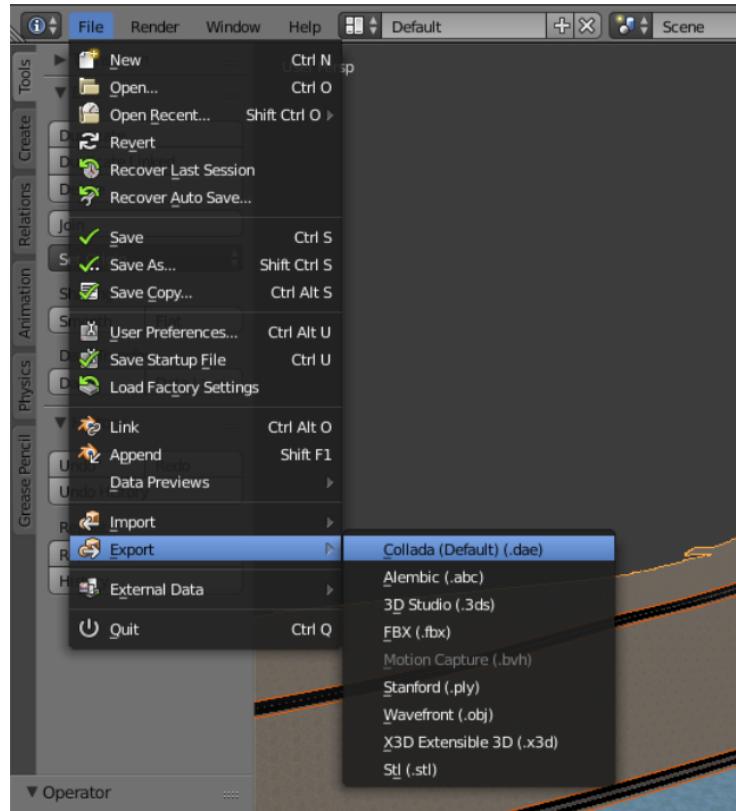


Figura 4.18: Desplegable para exportar ficheros desde Blender.

Al seguir los pasos aparecemos en otra ventana de Blender donde nos pregunta la ruta donde exportar el archivo y el nombre del fichero, entre otros parámetros. Especialmente importantes son los que se aprecian en la Figura 4.19. Si no se activan las casillas adecuadas se podrían producir errores en la exportación. Por ejemplo, dentro de *Export Data Options* es necesario marcar tanto *Apply Modifiers* como *Selection Only*. La primera casilla sirve para no perder los modificadores aplicados a los objetos del escenario, como la curva o la rejilla aplicadas al segmento del circuito (*Figura 4.10*). De no marcarlo podrían aparecer una fila de segmentos rectos y planos del circuito, o un circuito plano sobre las elevaciones, o un único segmento sin ninguno de los modificadores aplicados.

El segundo es para exportar sólo los objetos seleccionados al hacer click en exportar. De esta manera podemos ayudarnos de múltiples objetos y mantenerlos en la escena para futuras modificaciones, o editar varios objetos a la vez en la misma escena pero exportarlos individualmente. De no marcar la casilla se exportarían todos los elementos de la escena.

Dentro de *Texture Options* se encuentran las opciones relacionadas con las texturas. Aquí es necesario marcar las tres primeras casillas, *Only Selected UV Map*, *Include UV Textures*, *Include Material Textures* y *Copy*. Marcando la primera casilla se exportarán sólo las texturas de los objetos seleccionados, en lugar de todas las de la escena. La segunda y

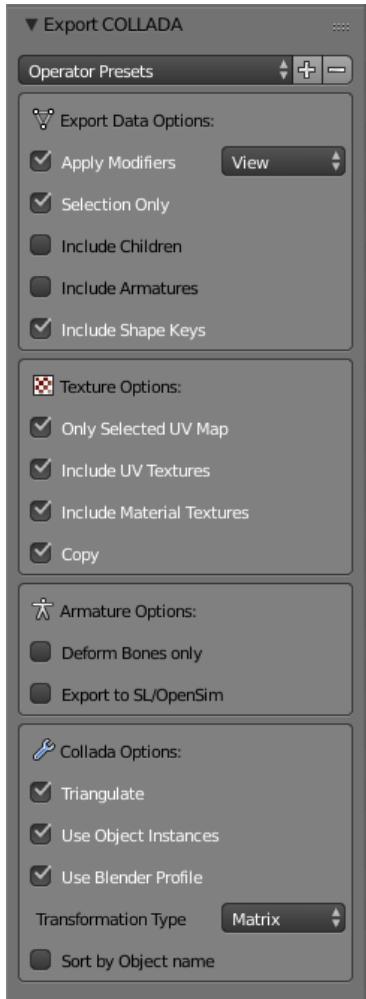


Figura 4.19: Opciones de exportación de Collada.

tercera casilla permiten exportar las texturas UV y los materiales usados respectivamente, de esta forma se mantienen las texturas que se veían en el modo de renderizado. Si simplemente asignamos una textura y no seguimos los pasos descritos anteriormente, en este paso Blender no las reconocería y no las exportaría junto con el resto del circuito, obteniendo así una figura sin imágenes en su superficie.

La última casilla copia los archivos de las texturas a la carpeta de destino de la exportación, y los enlaza dentro del fichero .dae. De esta forma se pueden copiar los archivos resultantes a cualquier otro dispositivo sin perder ningún dato por el camino o hacer más configuración manual. Las secciones de *Armature Options* y *Collada Options* no son relevantes en nuestro caso, y con dejar las casillas marcadas por defecto es suficiente.

Una vez tenemos exportado el circuito a un fichero .dae, creamos el fichero .sdf que importaremos en el mundo de Gazebo. El fichero finalizado tendrá el siguiente contenido:

```
1 <?xml version='1.0'?>
```

```

2 <sdf version="1.4">
3 <model name="monaco">
4   <static>true</ static>
5   <link name="monaco">
6     <pose>0 0 0 0 0 0</ pose>
7     <collision name="collision">
8       <geometry>
9         <mesh>
10        <scale>15 15 15</ scale>
11        <uri>model://monaco/meshes/CircuitoMonaco.dae</ uri>
12      </mesh>
13    </geometry>
14  </collision>
15  <visual name="visual">
16    <geometry>
17      <mesh>
18        <scale>15 15 15</ scale>
19        <uri>model://monaco/meshes/CircuitoMonaco.dae</ uri>
20      </mesh>
21    </geometry>
22  </visual>
23 </link>
24 </model>
25 </sdf>

```

Dentro de este fichero podemos diferenciar claramente el campo *link* entre las líneas 5 y 23. En este campo se importan los archivos y configuraciones que componen el modelo sdf. Podemos ver tres campos: *pose*, *collision* y *visual*. El primero define la posición y la orientación del objeto. En nuestro caso, como estamos creando el modelo del mundo donde se situarán los demás objetos, podemos dejar cualquiera con tal de que esté minimamente centrado en el mundo. El segundo define las barreras de colisión con los demás objetos del mundo. Es importante dado que, de no incluirlo, Gazebo trataría el objeto como si de un holograma se tratase, permitiendo que cualquier otro objeto lo atravesase sin problemas. Y el tercero define la parte que se ve del objeto. Si no lo definimos obtendremos un circuito invisible. Ambos campos tienen definidos unos valores de *scale*. Esto es así para mantener una cohesión con los demás mundos creados en JdeRobot y con los vehículos existentes. Es importante también el campo de la línea 4, *static*, con valor *true*. Este campo fuerza al circuito a quedarse inmóvil en el espacio de Gazebo y actuar como suelo para los coches. De no definirlo o no asignarle el valor *true*, obtendremos un circuito que al comenzar la simulación de Gazebo caerá al vacío, junto con los demás objetos.

Para cumplir con el estándar de SDF es necesario también crear un fichero de configuración como este:

```

1 <?xml version="1.0"?>
2   <model>
3     <name>Monaco flat</name>
4     <version>1.0</version>
5     <sdf version='1.5'>model.sdf</sdf>
6
7     <author>
8       <name>Alvaro Villamil</name>
9     </author>
10
11    <description>
12      The F1 Monaco track without elevations .
13    </description>
14 </model>
```

De este fichero, de extensión .config, SDF extrae información como el nombre del modelo, la versión usada para crearlo, la descripción del mismo, el archivo al cual aplicar esta configuración o el autor.

A continuación estructuramos los ficheros para que mantengan coherencia con el resto de modelos de JdeRobot. De tal forma creamos una carpeta con el nombre del modelo, en este caso *monaco*, y dentro de ella situamos los archivos *model.config* y *model.sdf* y una carpeta que llamamos *meshes*. Dentro de esta carpeta guardamos el archivo *CircuitoMonaco.dae* y todas las texturas que necesita nuestro modelo, tal y como las exportó Blender.

El siguiente paso es crear el fichero *monaco.world*, el cual pasaremos como argumento a Gazebo al lanzarlo para que cargue nuestro mundo. Para decirle a Gazebo que queremos cargar nuestro circuito creamos el campo *include* de las líneas 7 a 10. De esta manera le decimos que cargue el modelo *monaco* y lo coloque en una posición en el mundo. También creamos un sol estándar de Gazebo en otro campo *include*. Al probar a lanzar Gazebo con esta configuración vemos que la escena queda muy oscura y decidimos añadir dos *light*, dos luces direccionales que además aportarán sombras y dinamismo al mundo. Para no crear una carga excesiva de trabajo de simulación hacemos que ambas luces sean idénticas, potenciando el efecto sin crear varias sombras ni reflejos innecesarios. El contenido final de dicho fichero es éste:

```

1 <?xml version="1.0"?>
2 <sdf version="1.4">
3   <world name="default">
4     <include>
5       <uri>model://sun</uri>
6     </include>
7     <include>
8       <uri>model://monaco</uri>
9       <pose>0 0 0 0 0 0</pose>
10    </include>
11
12    <light name='user_directional_light_0' type='directional'>
13      <pose frame='>0.467439 -3.3788 2.45574 0.487341 -0 0</pose>
14      <diffuse>0.5 0.5 0.5 1</diffuse>
15      <specular>0.1 0.1 0.1 1</specular>
16      <direction>0.1 0.1 -0.9</direction>
17      <attenuation>
18        <range>20</range>
19        <constant>0.5</constant>
20        <linear>0.01</linear>
21        <quadratic>0.001</quadratic>
22      </attenuation>
23      <cast_shadows>1</cast_shadows>
24    </light>
25    <light name='user_directional_light_1' type='directional'>
26      <pose frame='>0.467439 -3.3788 2.45574 0.487341 -0 0</pose>
27      <diffuse>0.5 0.5 0.5 1</diffuse>
28      <specular>0.1 0.1 0.1 1</specular>
29      <direction>0.1 0.1 -0.9</direction>
30      <attenuation>
31        <range>20</range>
32        <constant>0.5</constant>
33        <linear>0.01</linear>
34        <quadratic>0.001</quadratic>
35      </attenuation>
36      <cast_shadows>1</cast_shadows>
37    </light>
38
39  </world>
40 </sdf>
```

Para incorporarlo al conjunto de mundos y modelos de JdeRobot, primero

comprobamos en nuestro ordenador que funciona correctamente. Para ello copiamos la carpeta *monaco* con los archivos correspondientes al modelo dentro de la carpeta *JdeRobot/src/drivers/gazeboserver/models*, y el fichero *monaco.world* dentro de la carpeta *JdeRobot/src/drivers/gazeboserver/worlds*. Al lanzar Gazebo con nuestro mundo como argumento no da ningún error y aparece el circuito correctamente. Una vez probado que todo funciona repetimos el proceso para cada circuito, obteniendo cuatro modelos y cuatro mundos, los correspondientes al circuito plano, al circuito con elevaciones, al circuito plano con la línea roja y al circuito con elevaciones con la línea roja.

Sin embargo, las precauciones tomadas no han sido suficientes y los circuitos de Mónaco con elevaciones llevan una gran carga gráfica que hace que la simulación se mueva lenta. Para disminuir esta carga recortamos parte de la rejilla que sirve de fondo, reduciendo la zona del mar y la zona alta detrás del circuito. Son zonas lejanas al trazado que no van a disminuir la apariencia general, pero si a mejorar el rendimiento computacional. Una vez hecho este cambio comprobamos que la simulación en Gazebo se mueve con soltura y nos permite añadir objetos sin afectar al rendimiento.



Figura 4.20: Coches recorriendo el circuito de Mónaco.

Cuando ya hemos comprobado que con ninguno de los cuatro mundos se producen errores procedemos a hacer un *pull* al repositorio oficial de JdeRobot<sup>5</sup> en GitHub. De esta forma logramos incorporar nuestros progresos al código oficial de JdeRobot, y que cualquiera que se descargue esta plataforma pueda disfrutar de los mundos que hemos

---

<sup>5</sup><https://github.com/JdeRobot/JdeRobot>

creado. En la figura 4.20 podemos observar el resultado en Gazebo con coches circulando por el circuito como si de una práctica académica se tratase. También comprobamos que la fluidez del mundo al aumentar la carga de trabajo sigue siendo satisfactoria, y va a permitir realizar diferentes tipos de prácticas.



# Capítulo 5

## Brazo robótico

En este capítulo vamos a estudiar el comportamiento y la estructura de un brazo robótico para luego crear un controlador de bajo nivel, un teleoperador, para él. Esta práctica supone la primera incursión en JdeRobot en el ámbito de los brazos robóticos y prepara el terreno para posteriores prácticas de planificación de movimientos articulados con este tipo de robots.

### 5.1. Brazos robóticos existentes en el entorno ROS

Dado que el objetivo no es crear un brazo, sino trabajar sobre uno ya diseñado, comenzamos buscando uno sobre el cual realizar el estudio. Debido a un cambio en la distribución de los paquetes de ROS entre las versiones Indigo y Jade, la mayoría de brazos manipuladores encontrados son antiguos y dan muchos problemas de instalación y configuración en las versiones más modernas. A partir de Indigo, los paquetes que forman ROS Industrial se han separado del núcleo de ROS, no están mantenidos por ROS, por lo que el desarrollo y la corrección de errores es más lenta. Puesto que queremos realizar el trabajo sobre ROS Kinetic y Gazebo 7 necesitamos encontrar un brazo que funcione bajo estas versiones. Tanteamos el uso de varios modelos:

- PR2: Se trata de un robot con apariencia de androide (*Figura 5.1*). Se compone de una base con ruedas, dos brazos y una cabeza con diferentes sensores visuales, siendo los brazos donde centraríamos nuestra atención. Cuando miramos su página de ROS<sup>1</sup> observamos que no da soporte más allá de Indigo, lo cual puede suponer un problema.

---

<sup>1</sup><http://wiki.ros.org/Robots/PR2>

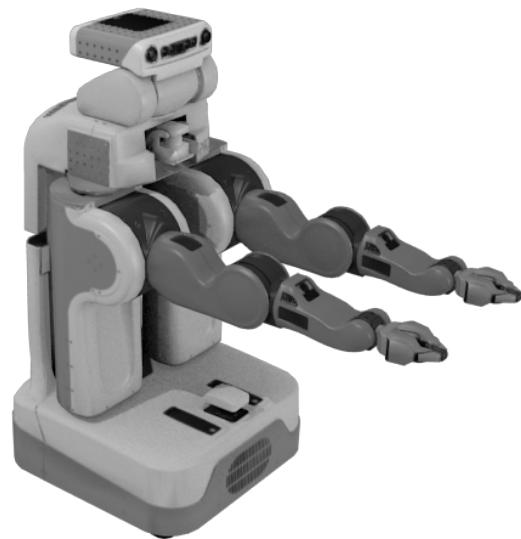


Figura 5.1: Robot PR2.

- kuka: La empresa Kuka<sup>2</sup> fabrica diversos brazos mecánicos y robots industriales. Hay varios modelos disponibles para su uso con Gazebo y ROS, y todos tienen en común el brazo que monta el robot de la Figura 5.2, aunque la plataforma donde se apoya el brazo puede ser diferente o no estar. En su página de ROS<sup>3</sup> vemos que soporta las versiones de Indigo y Kinetic, pero mediante la instalación del paquete externo ROS-Industrial<sup>4</sup>.



Figura 5.2: Robot kuka.

---

<sup>2</sup><https://www.kuka.com/>

<sup>3</sup><http://wiki.ros.org/kuka>

<sup>4</sup><http://wiki.ros.org/Industrial>

- ur10: De la empresa Universal Robots<sup>5</sup>. Es un brazo simple (*Figura 5.3*), al igual que sus hermanos el ur3 y el ur5. La diferencia entre éstos no va más allá del tamaño y fuerza del robot. En su página de ROS<sup>6</sup> observamos que no da soporte más allá de Indigo, lo cual puede suponer un problema.



*Figura 5.3: Robot ur10.*

Probamos los brazos bajo la última versión de Gazebo y de ROS recurriendo a la extensa comunidad de ROS en busca de soluciones para incorporar los brazos a la última versión. Algunas de las soluciones propuestas y probadas por nosotros son la instalación manual desde fuente, la compilación manual del código fuente, paquetes alternativos de usuarios que han resuelto los problemas de compatibilidad o interfaces de usuarios para que el robot se comunique con las librerías actuales. Después de probar muchas de estas soluciones, y de intentar solucionar los problemas por nuestra cuenta, vemos que ninguno de ellos llega a funcionar de forma correcta bajo nuestros requisitos.

Finalmente recurrimos a los escenarios de ARIAC (*Sección 3.5*) para desarrollar nuestro trabajo. Como podemos ver en la Figura 3.2 se trata de un escenario industrial, con un brazo ur10 situado sobre un carril que le permite desplazarse. Nos centramos en esta parte del escenario y del código, estudiando cómo está construido y cómo funciona. Al comprenderlo mejor podemos ver que sigue un esquema como el de la figura 5.4, similar al

---

<sup>5</sup><https://www.universal-robots.com/es/>

<sup>6</sup>[http://wiki.ros.org/ur\\_gazebo](http://wiki.ros.org/ur_gazebo)

de las prácticas de JdeRobot-Academy. Por un lado, está el mundo de Gazebo con todos los componentes del escenario y el brazo. Por otro, el código que desarrollaremos para crear el teleoperador, y entre ellos la interfaz de comunicación de ROS. Las flechas verdes indican los elementos de ROS y de ARIAC que hemos necesitado para comunicarnos con el brazo, y la flecha roja señala la parte creada como consecuencia de este trabajo. Es de gran ayuda para comprender el funcionamiento del mundo y del brazo la página de documentación de ARIAC[10] donde se detallan las interfaces de comunicación de los elementos del escenario.

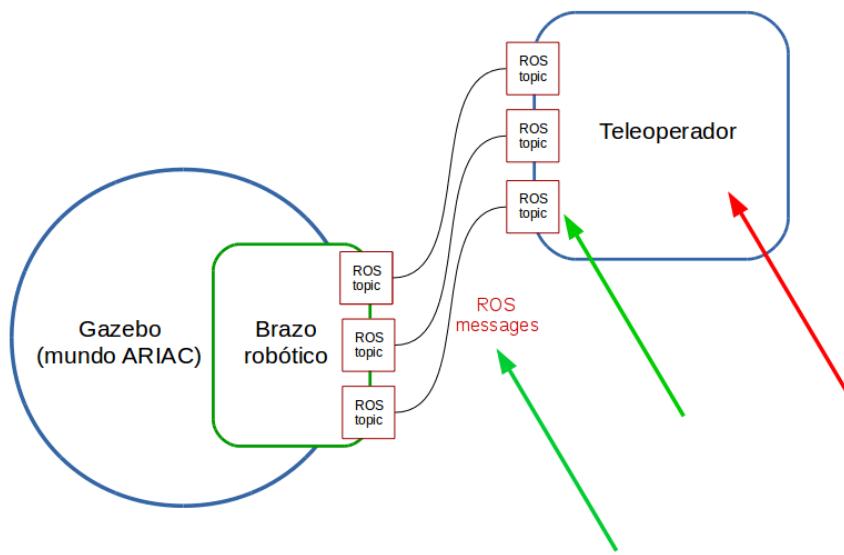


Figura 5.4: Esquema de los componentes de ARIAC centrándonos en el brazo.

Para poder avanzar y comenzar a probar cosas sobre el funcionamiento del brazo, necesitamos comprender mejor cómo funciona ROS.

## 5.2. Hablando ROS con un brazo robótico ARIAC

### 5.2.1. Nodos y topics

Para aprender el funcionamiento de ROS vamos a la documentación oficial[12]. Una vez allí nos damos cuenta de que los conceptos básicos que conocer para entender ROS son los *topics*, los *nodes* y los *messages*.

Los *nodes* son los procesos ejecutables, en nuestro caso escritos en Python, que se combinan entre sí siguiendo un esquema de grafo. Se comunican unos con otros mediante *topics*, servicios y acciones. Está pensado para que cada tarea dentro del robot la ejecute un nodo, facilitando APIs y canales de comunicación que hacen que el conjunto de código sea

sencillo de depurar y de utilizar. ROS facilita comandos de terminal como *rosnode list*, que muestra una lista de los nodos activos. Estos comandos son de gran ayuda para entender el funcionamiento del brazo.

Los *topics* son *buses* diferenciables mediante los cuales los nodos intercambian *ROS messages* o mensajes. Normalmente los *topics* y los nodos no se preocupan del destinatario de los mensajes. Los nodos se suscriben a los *topics*, y pueden publicar información en el *topic* haciendo *publish* o recibir información del *topic* haciendo *subscribe*. Por ejemplo un sensor de temperatura o de proximidad publica, mientras que un algoritmo de toma de decisiones o de procesado de datos se subscribe. Son canales de comunicación unidireccionales. Utilizan “tipado fuerte” mediante el tipo de *ROS message* que transmiten, de forma que un tipo de mensaje erróneo no se puede transmitir por el *topic* y no producirá fallos en otros nodos. ROS facilita comandos de terminal como *rostopic list* y *rostopic echo /topic\_name*. El primero muestra una lista con los *topics* activos, y el segundo permite ver el flujo de mensajes a través del *topic* deseado. Ambos son una ayuda inestimable para entender, y más tarde replicar, la comunicación con el brazo.

Los *servicios* son canales bidireccionales de comunicación entre nodos. Esto quiere decir que cuando se envía un mensaje, el nodo destino debe enviar otro de respuesta, o la comunicación no tiene éxito. Las *acciones* son un método de comunicación mas directo, como si de una llamada a procedimiento se tratase. Ninguno de estos medios son usados para enviar órdenes al brazo, por lo que no profundizamos en ellos.

Los *ROS messages* contienen la información que los nodos se envían mediante los *topics*. Se trata de estructuras de datos con campos fijos, que soportan tanto tipos de datos primitivos (enteros, *floats*, *boleanos*, etc) como arrays de estos tipos de datos. Estas estructuras están predefinidas en la mayoría de casos comunes, pero se pueden definir nuevas usando archivos en formato .msg donde se especifique la estructura de datos del mensaje

Una vez conocemos los elementos de comunicación probamos cómo se produce, creando un nodo *publisher* y otro *subscriber* y haciendo que hablen entre sí. Mediante los diferentes comandos de terminal que proporciona ROS comprobamos el correcto funcionamiento de los mismos.

### 5.2.2. Nodos, *topics* y mensajes de un brazo robótico ARIAC

Una vez entendemos el funcionamiento de ROS lanzamos el mundo de ARIAC. A través de los comandos visualizamos la lista de nodos y *topics* disponibles. Nos servimos de una

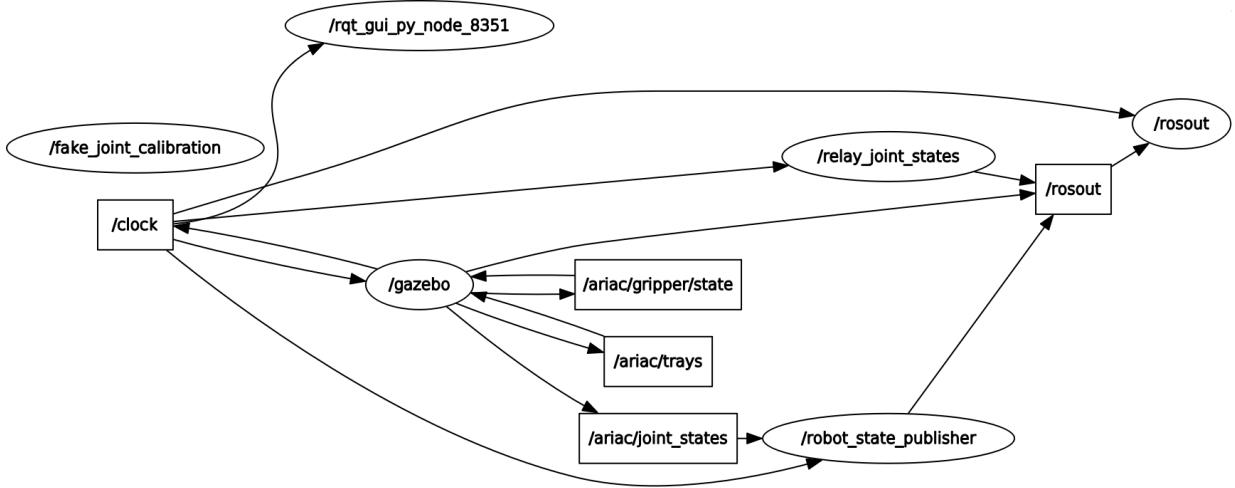


Figura 5.5: Grafo de los nodes y topics de ARIAC.

herramienta de ROS para obtener un grafo con la relación entre todos ellos y tener una idea clara del funcionamiento y la relación entre los elementos del brazo. Para ello ejecutamos en la terminal `rosrun rqt_graph rqt_graph` y obtenemos la Figura 5.5. En dicha figura, los elementos redondos son nodos y los cuadrados son *topics*, y las flechas establecen la relación entre ellos. Si la flecha apunta hacia un *node* desde un *topic* quiere decir que el nodo está suscrito a ese *topic*. Si por el contrario la flecha va desde un *node* a un *topic* quiere decir que el nodo publica en ese *topic*.

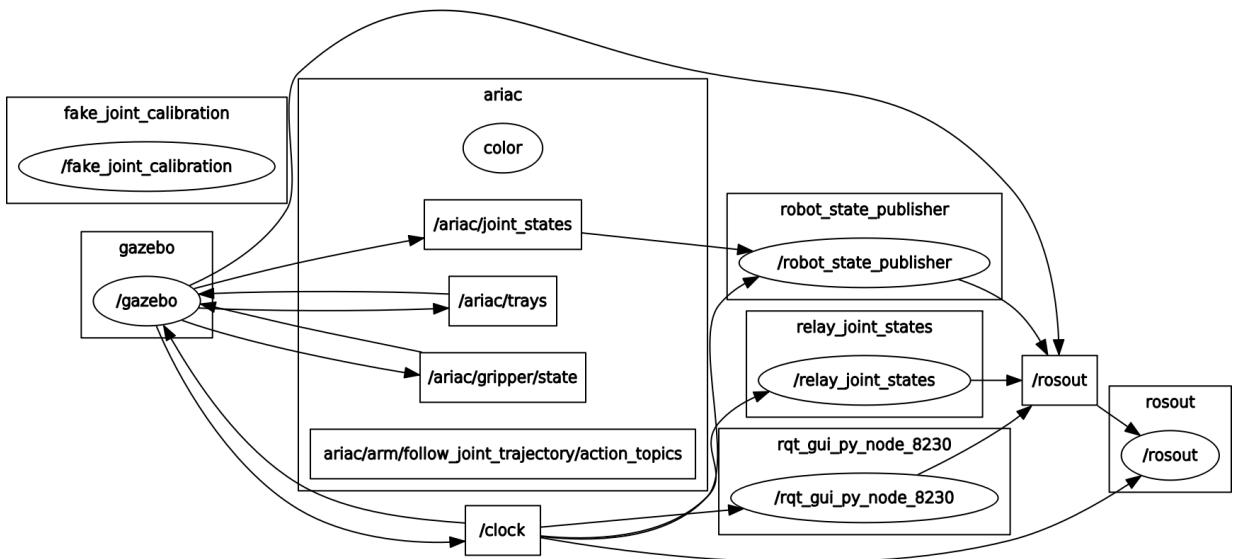


Figura 5.6: Grafo de los nodes y topics de ARIAC.

Aunque ya nos da una idea muy clara de la organización de los nodos y *topics*, la

herramienta *rqt\_graph* nos permite agruparlos por nombres. Dado que hay varios *topics* cuyo nombre es “/ariac/nombre”, *rqt* incluye a todos los que tienen ese formato en el nombre en la caja ARIAC. De esta forma la caja ariac hace referencia a todos los nodos y *topics* que comparten el nombre ariac. Si hacemos esto obtenemos el grafo de la figura 5.6.

En esa figura podemos diferenciar varios bloques importantes. En el centro del grafo se sitúan los *topics* y nodos referentes al mundo ARIAC, A su izquierda se encuentra el nodo principal de Gazebo. Encima de éste y a la derecha del bloque ariac hay varios grupos que sirven de apoyo para el entorno: *fake\_joint\_calibration*, *robot\_state\_publisher* y *relay\_joint\_states*. Debajo de ariac se encuentra el topic “/clock”, que sirve de referencia para todos los elementos de la simulación. A la derecha de éste está el nodo de *rqt* que nos permite obtener esta imagen. En la parte derecha del grafo hay diversos nodos y *topics* de *rosout*. Se trata de la consola de *log* de ROS, donde se anotan los sucesos o errores de los diferentes procesos ejecutados.

Una vez tenemos este esquema en la cabeza acudimos tanto al código de ARIAC como a su documentación para ver qué hace cada nodo y cada topic, qué partes controlan el brazo y cómo podemos comunicarnos con ellas. De esta forma descubrimos dos *topics* clave para poder controlar el brazo:

- /ariac/joint\_states: En este *topic* se publica la información relativa las posiciones de las articulaciones del brazo. Por este *topic* se envían mensajes de tipo *sensor-msgs/JointState Message*, que veremos más adelante. Nos subscribiremos a este *topic* para conocer la posición inicial del brazo y de sus articulaciones.
- /ariac/arm/command: Por este *topic* se envía al brazo la posición a la que quieres moverlo mediante mensajes del tipo *trajectory-msgs/JointTrajectory Message*, que veremos más adelante. Necesitamos publicar en este *topic* para enviar órdenes al brazo.

Los demás *topics* tienen otras funciones que no necesitamos para conseguir nuestro objetivo.

Los mensajes descritos anteriormente tienen la siguiente estructura:

- *sensor-msgs/JointState Message*:

```

1 std_msgs/Header header
2 string [] name
3 float64 [] position
4 float64 [] velocity

```

```
5     float64 [] effort
```

- *trajectory-msgs/JointTrajectory Message:*

```
1     std_msgs/Header header
2     string [] joint_names
3     trajectory_msgs/JointTrajectoryPoint [] points
```

Ambos mensajes comienzan con un *Header* de tipo *std\_msg/Header*. Este es un tipo de mensaje estándar de ROS, que necesitan todos los mensajes para que pueda establecerse la comunicación, y se crea automáticamente al enviar el mensaje. No necesitamos crearlo para poder enviar mensajes y no necesitamos extraerlo al recibirlos, por lo que no nos preocupamos por él.

El primer tipo de mensaje lo componen varios *arrays* de datos primitivos. El primero de ellos es un *array* de *strings*. Los *strings* son cadenas de caracteres, como palabras o frases. Este campo lo componen los nombres de las articulaciones del brazo, que son:

- elbow\_joint : Ésta es la articulación del codo, la del medio del brazo.
- linear\_arm\_actuator\_joint : Ésta no es una articulación propiamente dicha, sino que se refiere a la posición del brazo en el carril sobre el que está situado.
- shoulder\_lift\_joint : Ésta es la articulación del hombro, es decir, la más cercana a su base. En concreto controla la elevación del brazo. Realiza giros sobre un imaginario eje y.
- shoulder\_pan\_joint : Ésta es la articulación del hombro, es decir, la más cercana a su base. En concreto controla la orientación del brazo y nos permite girarlo. Realiza giros sobre un imaginario eje z.
- wrist\_1\_joint : Ésta es la articulación de la muñeca, la más alejada de la base. En concreto controla la elevación de la muñeca y nos permite subirla y bajarla. Realiza giros sobre un imaginario eje y.
- wrist\_2\_joint : Ésta es la articulación de la muñeca, la más alejada de la base. En concreto controla el giro de la muñeca y nos permite rotarla verticalmente. Realiza giros sobre un imaginario eje z.
- wrist\_3\_joint : Ésta es la articulación de la muñeca, la más alejada de la base. En concreto controla el giro de la mano y nos permite rotar la muñeca horizontalmente. Realiza giros sobre un imaginario eje x.

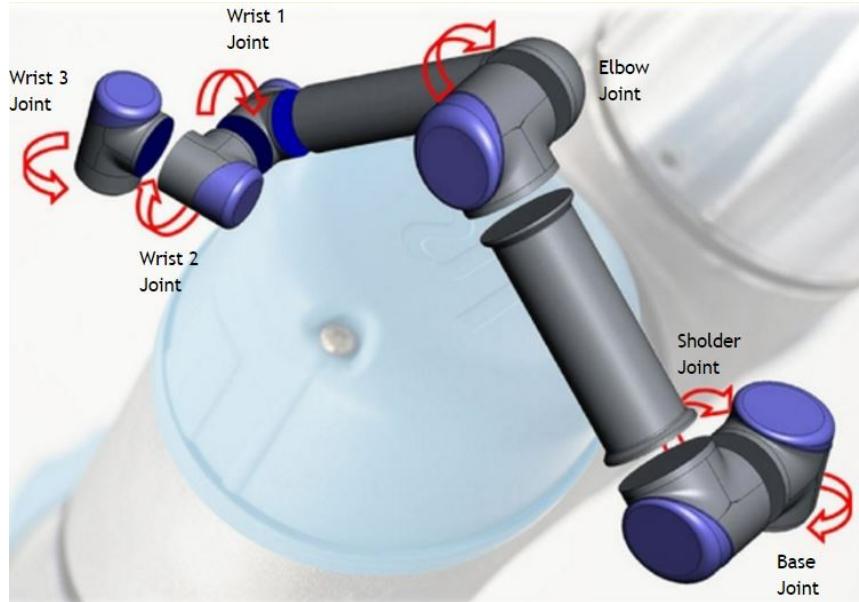


Figura 5.7: Detalle de las articulaciones del brazo.

- `vacuum_gripper_joint` : Esta articulación hace referencia a una pinza de vacío que se puede acoplar a la articulación de la muñeca del brazo, pero en nuestro caso no la incorpora y no necesitamos asignarle valores

En la Figura 5.7 podemos situar visualmente las articulaciones en el brazo, con la diferencia de que la articulación *Base Joint* de la imagen en ARIAC se llama *shoulder\_pan\_joint*. Todas estas articulaciones tienen definida su posición en radianes y unos límites de giro entre 6,28 y -6,28 en su mayoría. Esto quiere decir que cada articulación del brazo puede realizar dos giros completos si fuese necesario, pero los choques con los elementos de escenario limitan los movimientos.

Los siguientes campos del mensaje son *arrays* de *floats* (números) de posiciones, velocidades y esfuerzos. Dan información de la posición de cada articulación, la velocidad a la que se mueven, y el esfuerzo o fuerza que poseen en el momento de envío del mensaje, por lo que son arrays de ocho posiciones y en cada una está la información relativa a la articulación en la posición homóloga en el array de nombres. A nosotros nos interesa el primer *array* para conocer la posición inicial de las articulaciones.

Para ver mejor la estructura del mensaje lanzamos el mundo de ARIAC y ejecutamos el comando de terminal `rostopic echo /ariac/joint_states`, obteniendo en texto en la terminal la estructura de datos de este mensaje:

```

1 ——
2 header:
3 seq: 265
4 stamp:
5 secs: 5
6 nsecs: 322000000
7 frame_id: ''
8 name: [ 'elbow_joint', 'linear_arm_actuator_joint', 'shoulder_lift_joint',
      'shoulder_pan_joint', 'wrist_1_joint', 'wrist_2_joint', '
      'wrist_3_joint', 'vacuum_gripper_joint' ]
9 position: [1.5072954794978815, 0.04169673392358113, -0.3793140712872205,
            3.217984505432054, 3.085834205511564, -1.6130778569148077,
            -0.00793595855568796, 0.0]
10 velocity: [0.027474972846918494, 0.007286219434794008,
              0.05461882849916373, 0.0316967471028743, 0.1957558318356749,
              -1.091011699832568, -8.054309845386587, 0.0]
11 effort: [150.0, -184.23945050318943, 330.0, 167.6704306272329, 0.0,
            5.601291580407757, -5.541011517572109, 0.0]
12 ——

```

El *header* o cabecera ocupa las primeras siete líneas, y refleja datos como el tiempo de simulación o el número de mensaje, ninguno de ellos relevante para nosotros. A continuación nos encontramos los cuatro *arrays*: *name* para los nombres de las articulaciones, *position* para las posiciones, *velocity* para las velocidades y *effort* para las fuerzas. Este mensaje es el que recibiremos y que deberemos procesar para obtener la posición inicial del brazo.

El segundo tipo de mensaje lo componen un *array* de *strings* y otro campo llamado *points* compuesto por otro tipo de mensaje, el tipo *trajectory-msgs/JointTrajectoryPoint*. El primer *array* hace referencia a los nombres de las articulaciones, pero en este mensaje se llama *joint\_names*. Como podemos ver a continuación, este tipo de mensaje está formado por *arrays* de números de forma muy similar a los del primer mensaje:

```

1 float64 [] positions
2 float64 [] velocities
3 float64 [] accelerations
4 float64 [] effort
5 duration time_from_start

```

Tenemos un *array* para las posiciones, otro para las velocidades, otro para las aceleraciones, otro para las fuerzas, y un último campo de tipo *duration* para especificar el número de segundos desde el inicio para ejecutar los movimientos. Nosotros usaremos tanto el *array*

de nombres como el de posiciones, así como el campo para establecer el tiempo inicial de la orden. Los demás campos no son necesarios para el correcto funcionamiento del brazo.

De la misma forma que con el mensaje anterior, arrancamos la simulación y ejecutamos en la terminal el comando `rostopic echo /ariac/arm/commander` para obtener en texto el contenido de los mensajes que pasan por este *topic*. Nos servimos de los tutoriales para dar órdenes sencillas a través de comandos de terminal y poder capturar el contenido de estos mensajes:

```

1 ——
2 header:
3 seq: 59
4 stamp:
5 secs: 0
6 nsecs: 0
7 frame_id: ''
8 joint_names: [ 'elbow_joint', 'linear_arm_actuator_joint', '
    shoulder_lift_joint', 'shoulder_pan_joint', 'wrist_1_joint', '
    wrist_2_joint', 'wrist_3_joint']
9 points:
10 —
11 positions: [1.510634135925831, 1.4199980429372933e-06,
    -1.1286545928274752, 3.140002031709801, 3.772089274964015,
    -1.5100101552695162, 4.0770707014914365e-06, 0.0]
12 velocities: []
13 accelerations: []
14 effort: []
15 time_from_start:
16 secs: 1
17 nsecs: 0
18 ——

```

Al igual que en el primer tipo de mensaje, la cabecera la componen las siete primeras líneas. Después nos encontramos con el *array* de nombres de articulaciones. A continuación nos encontramos con el campo *points*, es decir, el campo que contiene otro tipo de mensaje. Dentro podemos ver los cuatro arrays, y comprobamos que sólo el de posiciones está definido, los demás están en blanco y no son imprescindibles para dar órdenes al brazo. Por último podemos ver el campo *time\_from\_start*, el cual establece el tiempo en 1 segundo.

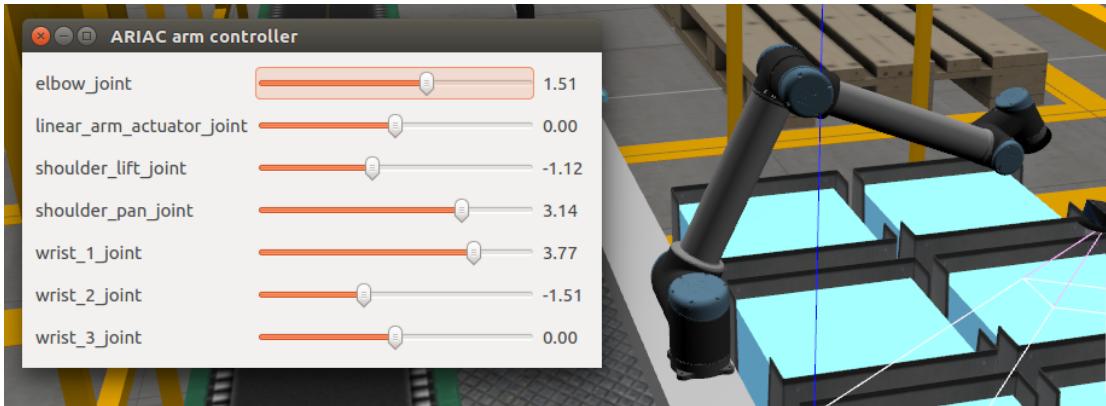


Figura 5.8: Interfaz gráfica del controlador del brazo y brazo en la posición inicial.

### 5.3. Teleoperador de un brazo robotizado

Una vez estudiados los mensajes y los *topics* que necesitamos para controlar el brazo comenzamos a implementar el controlador. Utilizando Python como lenguaje, creamos un archivo en el que vamos probando paso a paso la aplicación de los conceptos adquiridos, tanto de ROS como de ARIAC. Primero creamos un nodo al que llamaremos “Mando”. Luego hacemos que se subscriba al *topic* */ariac/joint\_states*, que descomponga los mensajes recibidos y que imprima por terminal los datos extraídos.

A continuación hacemos que el nodo publique en el *topic* */ariac/arm/command* una orden predefinida que hace que se mueva ligeramente, simplemente para comprobar que el envío de un mensaje bien construido a este *topic* consigue mover el brazo. Para ello debemos construir el mensaje de acuerdo a las especificaciones anteriormente descritas. Vemos que es necesario introducir el envío de mensajes en un bucle, ya que si enviamos uno y salimos, ROS cierra el nodo antes de que el mensaje llegue, con lo que no se recibe.

Una vez realizadas estas tareas pasamos a desarrollar la interfaz gráfica que nos permita controlar el brazo visualmente. Nos servimos de Qt para desarrollar la interfaz visual. Creamos unos deslizadores para cambiar los valores de posición de cada articulación por separado, consiguiendo de una forma heurística saber dónde están los límites de cada articulación y cuál es su posición respecto al total de su movimiento. También creamos unas etiquetas que muestran el valor numérico de la posición del deslizador, así como unas etiquetas con el nombre de la articulación que controla cada uno.

En lugar de utilizar la interfaz gráfica proporcionada por Qt para construir nuestra GUI la escribimos directamente en código utilizando las propiedades de auto-colocación de objetos. Qt nos permite insertar los elementos como si de una tabla se tratase y automáticamente los separa y les da un espacio suficiente para mostrarse completos.

También resulta más fácil programar después el comportamiento de la interfaz, ya que tenemos un mayor control sobre lo que creamos.

A la hora de crear el controlador agrupando todos los elementos, podemos organizar el código en base a si es relativo a la interfaz o a ROS, por lo que lo dividimos en dos carpetas dentro de la carpeta principal: *gui* y *ros\_manager*. Organizamos todo el código necesario en cinco ficheros:

- *main.py*: Este fichero es el encargado de lanzar los procesos relativos a la interfaz y a ROS. Es, por tanto, el ejecutable que invocamos para abrir el controlador. Se encuentra en la carpeta principal, y lanza los hilos de ejecución tanto de ROS como de la interfaz. Las partes más importantes del código de este fichero son:

```

1     ros_manager = RosManager()
2     (...)
3     myGUI = Window(ros_manager)
4     (...)
5     t = ThreadGUI(myGUI)
6     (...)
7     t.start()
```

En la primera línea creamos el objeto que contiene las funciones que se comunican con ROS. En la línea 3 y 5 inicializamos la ventana y el hilo donde se va a ejecutar, finalmente en la línea 7 arrancamos el hilo que gestiona la ventana gráfica.

- *gui.py*: En este fichero se encuentran definidos los elementos de la interfaz gráfica, así como el comportamiento de los mismos. Aquí creamos cada elemento, lo colocamos en la ventana, definimos los *callbacks* de los deslizadores y capturamos el cierre de la ventana para realizar un cierre ordenado. Los *callbacks* nos permiten capturar el movimiento de cada deslizador, actualizar la etiqueta correspondiente y enviar el nuevo valor al brazo. El cierre ordenado es necesario para poder cerrar el nodo de ROS y no dejar elementos corriendo en el sistema. A continuación se muestra parte del código utilizado, resumiendo aquellas partes repetitivas donde se repite la misma acción para varios elementos.

Creamos el objeto de Qt que creará el espacio para desplegar los elementos de la ventana:

```
1     mainLayout = QGridLayout()
```

Creamos las etiquetas de cada articulación y escribimos el texto que aparecerá en la ventana, en este caso el nombre real de cada articulación:

```

1     self.label1 = QLabel("elbow_joint")
2     (... )
3     self.label7 = QLabel("wrist_3_joint")

```

Creamos los deslizadores y asignamos los valores límite a cada uno. Como valor inicial preguntamos, a través de *ros\_manager*, la posición real de esa articulación y la redondeamos a dos decimales. En el caso de los deslizadores tiene truco, ya que no saben trabajar con valores decimales. Por eso, aunque los valores mostrados son los reales, trabajamos con ellos con los valores multiplicados por cien, de forma que manejan con números enteros:

```

1     self.slider1 = QSlider(Qt.Horizontal)
2     self.slider1.setMinimum(-628)
3     self.slider1.setMaximum(628)
4     self.slider1.setValue( float('%.2f' % (self.ros_manager.read_elbow()
5         ))*100)
6     (... )
7     self.slider7 = QSlider(Qt.Horizontal)
8     self.slider7.setMinimum(-628)
9     self.slider7.setMaximum(628)
10    self.slider7.setValue( float('%.2f' % (self.ros_manager.read_wrist_3
11        ))*100)

```

Creamos las etiquetas que muestran los valores de las articulaciones y, al igual que con los deslizadores, les damos el valor inicial real del brazo:

```

1     self.value1 = QLabel('%.2f' % (self.ros_manager.read_elbow()))
2     (... )
3     self.value7 = QLabel('%.2f' % (self.ros_manager.read_wrist_3()))

```

Añadimos uno a uno todos los elementos creados, tanto las etiquetas de los nombres como los deslizadores como las etiquetas de los valores, al elemento de Qt que los mostrará por pantalla. También les damos una ubicación mediante unas coordenadas. las coordenadas (0,0) corresponden a la primera posición, arriba a la izquierda. La posición (1,0) está situada debajo de la primera, mientras que la (0,1) está a la derecha, y así hasta el tamaño deseado. En nuestra ventana el último elemento se coloca en la posición (6,2):

```

1     mainLayout.addWidget(self.label1, 0, 0)
2     (... )
3     mainLayout.addWidget(self.label7, 6, 0)

```

```

4     mainLayout.addWidget( self.slider1 ,0 ,1)
5     (...)
6     mainLayout.addWidget( self.slider7 ,6 ,1)
7     mainLayout.addWidget( self.value1 ,0 ,2)
8     (...)
9     mainLayout.addWidget( self.value7 ,6 ,2)

```

Definimos la conexión de los *callbacks* entre cada deslizador y la función que procesará el movimiento de este:

```

1     self.slider1.valueChanged.connect( self.valuechange1 )
2     (...)
3     self.slider7.valueChanged.connect( self.valuechange7 )

```

Como ya hemos posicionado todos los elementos, damos la orden para que cree la ventana y coloque los elementos en el sitio que hemos asignado. Definimos un nombre para que lo muestre en la barra de título:

```

1     self.setLayout( mainLayout )
2     self.setWindowTitle("ARIAC arm controller")

```

Aquí se muestra una de las funciones a las que enlazan los *callbacks*. Lo que hace es coger el valor a donde hemos movido el deslizador, enviarlo (a través de *ros\_manager*) al brazo para que se mueva a esa posición, y cambiar la etiqueta de la ventana con el nuevo valor:

```

1     def valuechange1( self ):
2         num = self.slider1.value()
3         self.ros_manager.move_elbow( float( num )/100 )
4         self.value1.setText( str( float( num )/100 ) )

```

Con esta función controlamos el cierre. Al intentar cerrar la ventana, esta función pausa el cierre y manda la orden de parada al componente de comunicación con ROS para que pueda salir ordenadamente, y continúa con el cierre:

```

1     def closeEvent( self , event ):
2         self.ros_manager.stop()
3         event.accept()

```

- *threadGUI.py*: Este fichero crea un hilo de ejecución para los elementos de la interfaz gráfica. Dado que este elemento es común a todas las ventanas creadas con Qt, usamos el proporcionado en los tutoriales. Básicamente es un bucle en el que cada

cierto tiempo actualiza todos los elementos de la ventana con los valores que hayan cambiado.

- *ros.py*: En este fichero se encuentra el código relativo a ROS, el que gestiona el envío y recepción de mensajes. Aquí creamos el nodo, nos subscribimos al brazo, publicamos en él, gestionamos los cambios de posición de los deslizadores de la ventana y controlamos el cierre ordenado del nodo ROS. A continuación mostramos parte del código, resumiendo las partes repetitivas.

Con esta función se crean los parámetros iniciales del objeto que se comunicará con ROS. Primero se conecta con el topic de ROS donde publicará los mensajes para mover el brazo. Define el array con los nombres de las articulaciones y con las posiciones que enviará en los mensajes. Este array de posiciones es muy importante, ya que en él se escriben los valores de posición cada vez que se mueve un deslizador en la ventana, y de aquí se recogen cada vez que se envían al brazo. Después crea un *lock* o candado para evitar errores de concurrencia. Al haber varios hilos en ejecución, se puede dar el caso de que una función quiera escribir en un dato al mismo tiempo que otra quiere cogerlo, produciendo un error. Al “dar” el candado a una sola función, sólo esa puede tocar ese dato, las demás deben esperar a que acabe, evitando estos errores. Por último, llama a la función *start*:

```

1  def __init__(self):
2      self.pub = rospy.Publisher("/ariac/arm/command",
3                                JointTrajectory, queue_size=10)
4
5      self.arm_joint_names = [
6          'elbow_joint',
7          'linear_arm_actuator_joint',
8          'shoulder_lift_joint',
9          'shoulder_pan_joint',
10         'wrist_1_joint',
11         'wrist_2_joint',
12         'wrist_3_joint',
13     ]
14
15     self.position = [0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
16                      0.00]
17
18     self.lock = threading.Lock()
19     ...
20     self.start()
```

Esta es la función que se llama cuando se crea el objeto. Lo primero que hace es crear un nodo ROS. Después se subscribe al nodo que publica la información de los estados del brazo, recibe un mensaje y se lo pasa a la función *get\_msg* que interpreta la información contenida en él. Por último, arranca el hilo de ejecución que nos permitirá comunicarnos con el brazo:

```

1  def start(self):
2      rospy.init_node('Mando', anonymous=True)
3
4      sub = rospy.Subscriber("/ariac/joint_states", JointState, self
5          .get_msg)
6
7      self.thread.start()

```

Esta función se llama al recibir el primer mensaje, y obtiene la posición de cada articulación y la almacena para poder inicializar los valores de la ventana. Podemos ver cómo antes de escribir los valores “coge” el candado y después lo “suelta”:

```

1  def get_msg(self, msg):
2      self.lock.acquire()
3      self.position[0] = msg.position[0]
4
5      self.position[7] = msg.position[7]
6      self.lock.release()

```

Esta función se usa para enviar mensajes al brazo. Primero construye el mensaje según la estructura de arrays vista anteriormente en este mismo capítulo, y una vez construido lo publica en el *topic* al que se ha suscrito:

```

1  def send_msg(self):
2      msg = JointTrajectory()
3      msg.joint_names = self.arm_joint_names
4      point = JointTrajectoryPoint()
5      self.lock.acquire()
6      point.positions = self.position
7      self.lock.release()
8      point.time_from_start = rospy.Duration(1.0)
9      msg.points = [point]
10     if not rospy.is_shutdown():
11         self.pub.publish(msg)

```

La siguiente función es la responsable de que al mover un deslizador en la ventana se pueda enviar ese valor al brazo. Aunque sólo se muestra la de una articulación, las demás siguen el mismo esquema. Simplemente escribe el nuevo valor en el array de posiciones y llama a la función que envía el mensaje:

```

1  def move_elbow(self, pos):
2      self.lock.acquire()
3      self.position[0] = pos
4      self.lock.release()
5      self.send_msg()
```

La siguiente función, al contrario de la anterior, contesta a las peticiones de los elementos de la ventana con el valor de la posición pedida. Al igual que la anterior, sólo se muestra una ya que las demás son muy similares:

```

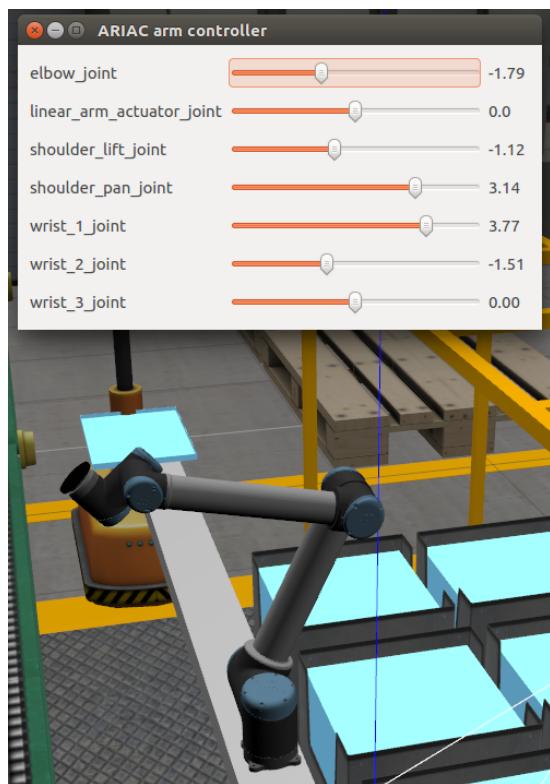
1  def read_elbow(self):
2      self.lock.acquire()
3      a = self.position[0]
4      self.lock.release()
5      return a
```

La última función es la encargada de realizar el cierre ordenado de los elementos de ROS. Además de imprimir un mensaje informando de que se ha iniciado la secuencia de cierre, apaga el nodo de ROS, ya que de no hacerlo se quedaría arrancado pero sin hacer nada hasta que se cierre el proceso principal de ROS:

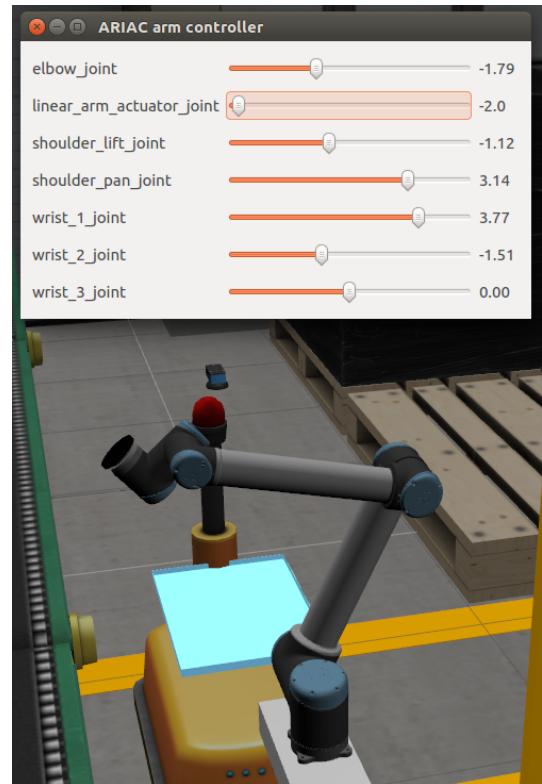
```

1  def stop(self):
2      print("Exiting the ARIAC arm controller... \n")
3      ...
4      self.pub.unregister()
5      rospy.signal_shutdown("Node Closed")
```

- *threadPublisher.py*: Este fichero crea un hilo de ejecución para los elementos de ROS. Dado que no es nuevo ni único para nuestro proyecto, reutilizamos uno de los usados en otros controladores de JdeRobot, simplemente cambiando el objeto de control del hilo. El funcionamiento básico de este hilo es crear un bucle en el que cada cierto tiempo se le envía al brazo un mensaje con la orden de movimiento. Aunque parezca extraño estar constantemente enviando mensajes, es la forma oficial, tanto en los tutoriales como en los ejemplos de la página de ROS, de comunicarse con los actuadores de los robots.



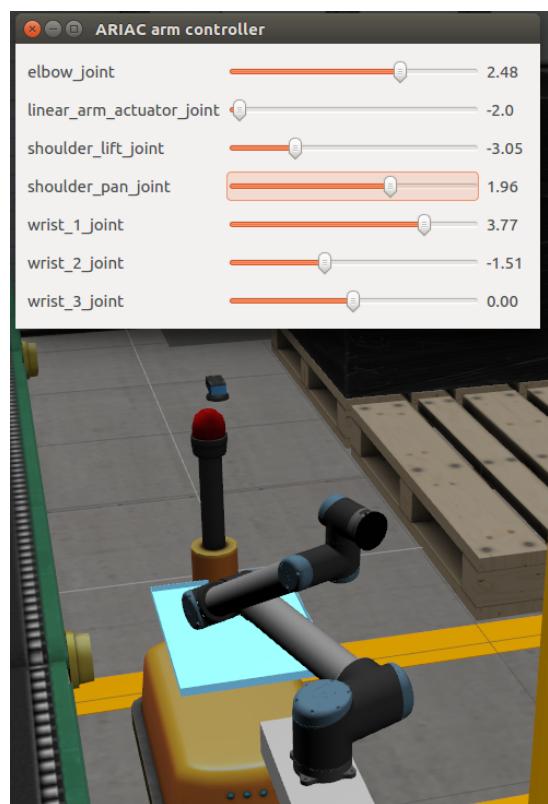
(a) elbow\_joint.



(b) linear\_arm\_actuator\_joint.

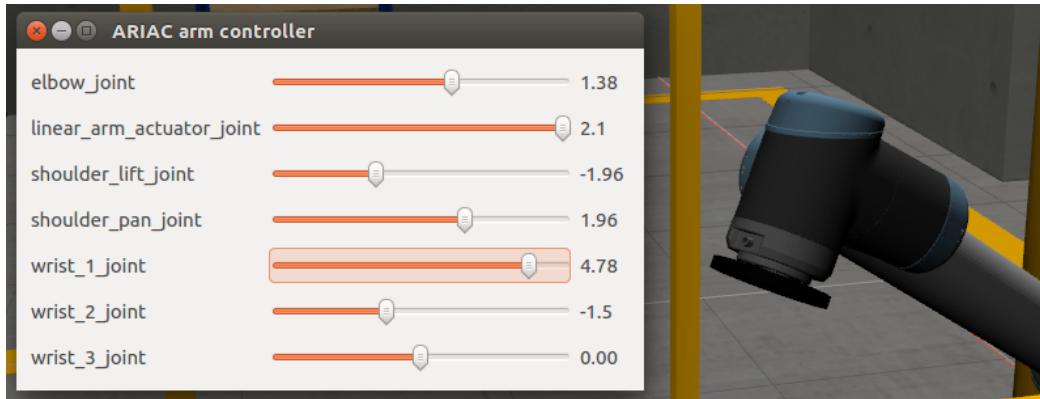


(c) shoulder\_lift\_joint.

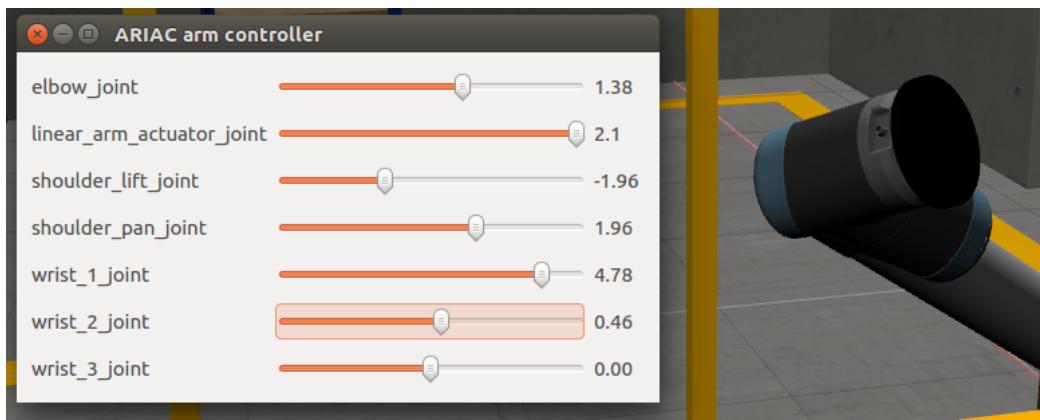


(d) shoulder\_pan\_joint.

Figura 5.9: Posiciones de las articulaciones del brazo después de: mover el codo (a); desplazarlo en el carril (b); mover el hombro (c); girar el hombro (d).



(a) wrist\_1\_joint.



(b) wrist\_2\_joint.



(c) wrist\_3\_joint.

Figura 5.10: Posiciones de las articulaciones de la muñeca después de: girarla en vertical (a); girarla en horizontal (b); girar la mano (c).

En la Figura 5.8 podemos ver la apariencia final del controlador del brazo con el que completamos nuestro objetivo inicial, así como la posición inicial del brazo al cargar el mundo de ARIAC.

Para ilustrar mejor los movimientos del brazo hemos realizado capturas de pantalla en las que se ilustran los grados de libertad del brazo. Se puede ver también el teleoperador, con el deslizador responsable del movimiento de la articulación resaltado. Si nos fijamos en el valor del deslizador que controla esa articulación veremos que es diferente, y se corresponde con la posición del brazo de la imagen. Dado que son siete grados de libertad en total, los hemos repartido en dos figuras.

En la figura 5.9 podemos ver los movimientos de las articulaciones del brazo, sin la muñeca, partiendo de la posición inicial de la figura 5.8. En la primera imagen mostramos el movimiento del codo. En la segunda desplazamos el brazo hasta el final del carril. En la tercera movemos el hombro, y recolocamos el codo para no colisionar con otros elementos del escenario. En la última giramos el hombro.

En la figura 5.10 podemos ver los movimientos de las articulaciones de la muñeca. En la primera imagen vemos como se ha inclinado hacia delante al mover la primera articulación. Al mover la segunda conseguimos girarla, como muestra la segunda imagen. En la tercera giramos la mano. Se puede apreciar por la muesca lateral que tiene el brazo cerca del disco de su extremo, que de la segunda a la tercera imagen se desplaza debido a este giro.

## 5.4. Manejo del brazo a través del planificador MoveIt

Al indagar y realizar pruebas sobre el funcionamiento de ARIAC y de ROS, vemos que existen herramientas que facilitan el control del brazo planificando movimientos. Una de ellas es MoveIt!, que se encarga de calcular trayectorias para facilitar los movimientos, lo que podríamos llamar un controlador de alto nivel. Esto lo hace conociendo el punto de partida del robot y recibiendo el punto de destino o un conjunto de puntos que le sirven de puntos de ruta a través de los cuales calcula la trayectoria. Es una herramienta compleja, ya que debe coordinar todas las articulaciones para que se desplacen hasta el punto final de una forma eficiente y sin chocar entre si.

Se sirve de herramientas como rViz para visualizar el robot, el punto de destino y la trayectoria, como podemos ver en la figura 5.11. En esta imagen podemos ver el brazo en su posición actual, al igual que se encuentra en Gazebo, y dos siluetas, una verde y otra naranja. La silueta verde identifica la posición de inicio, mientras que la naranja la posición

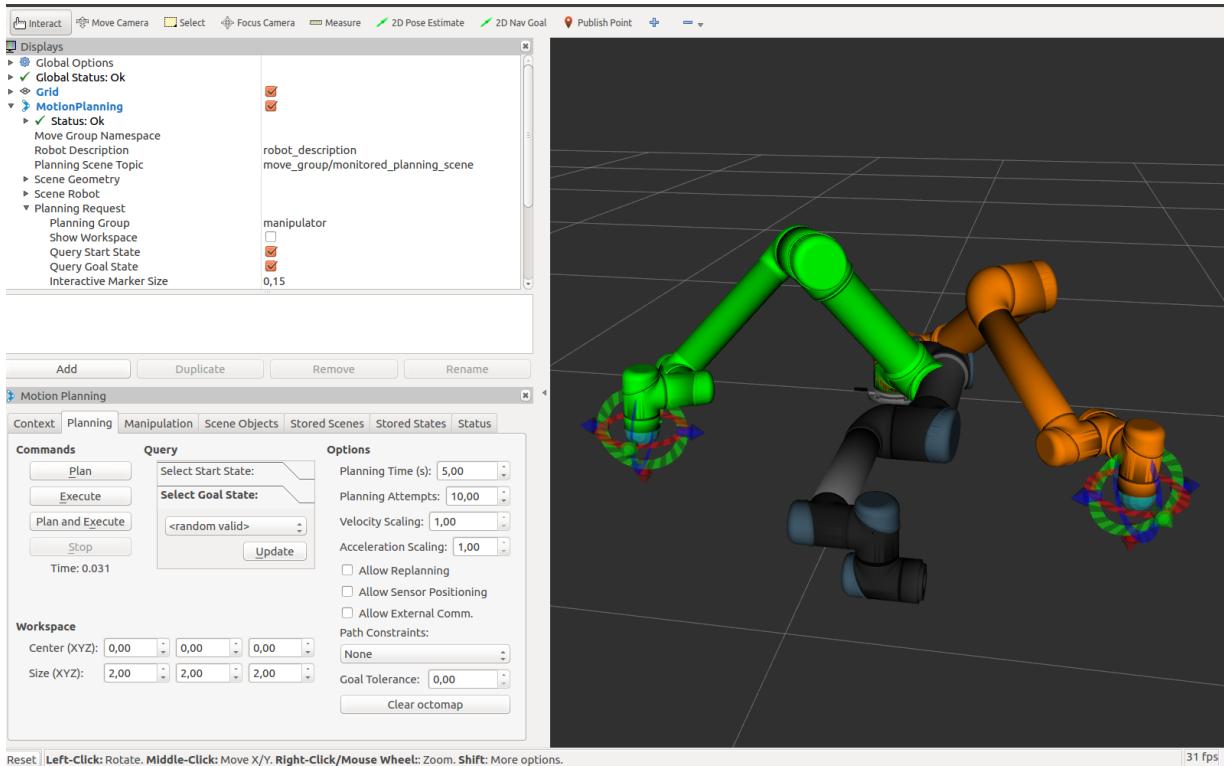


Figura 5.11: Herramienta rViz mostrando la trayectoria del brazo.

final. En la “mano” del robot podemos ver una bola rodeada de flechas y circunferencias de colores. Esa bola es el punto de referencia que MoveIt utiliza para calcular la trayectoria. Simplemente movemos esa bola por el espacio y la silueta se moverá para posicionar el brazo de forma que la bola esté donde nosotros queremos. Las flechas y círculos de colores son ayudas para mover y rotar la bola en torno a los ejes cartesianos.

Una vez alcanzada la posición final que queremos, MoveIt podrá planificar (y ejecutar si así lo deseamos) los movimientos necesarios para que el brazo alcance esa posición, sin que tengamos que preocuparnos por nada más.

Para conocer mejor el funcionamiento de esta herramienta utilizamos *rqt* al igual que hicimos para estudiar el funcionamiento de ARIAC, y obtenemos el grafo de la figura 5.12.

Como se puede apreciar es inmensamente grande, así que usamos las opciones de *rqt* para agrupar nodos y topics e intentar reducir su extensión, obteniendo la imagen de la figura 5.13. Sigue siendo un esquema enorme, lo cual nos da una idea de la complejidad de esta herramienta.

Si acudimos a la documentación oficial nos encontramos con una página<sup>7</sup> que muestra cómo interactuar con MoveIt. Es una interfaz muy detallada compuesta por mensajes,

<sup>7</sup>[http://wiki.ros.org/moveit\\_msgs](http://wiki.ros.org/moveit_msgs)

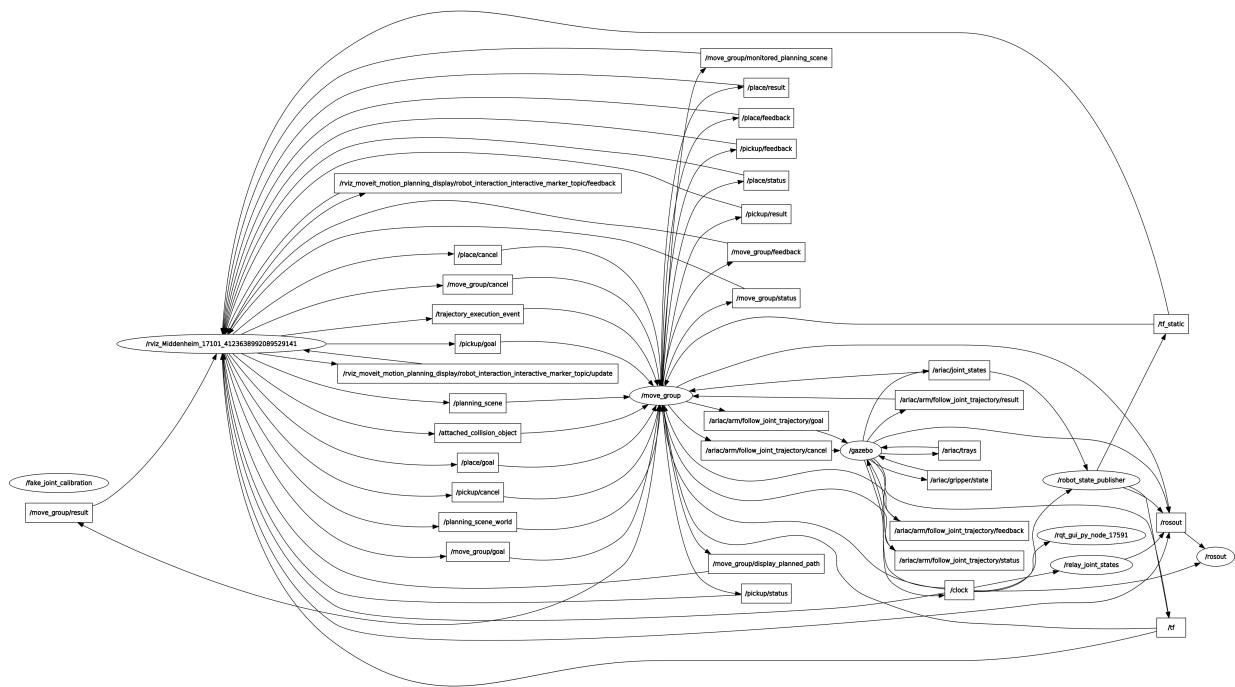


Figura 5.12: Grafo con la relación de nodos y topic de ARIAC, el brazo, MoveIt y rViz.

servicios y acciones de ROS. En la figura 5.14 se muestra una captura de pantalla de dicha página, con todos los elementos de la interfaz. Como podemos ver es una cantidad enorme de mensajes, cada uno con su propio tipo de estructura, que hace muy complejo y largo trabajar con esta herramienta.

A pesar del atractivo de esta herramienta y de sus posibilidades académicas descartamos seguir investigando en esta dirección, ya que consideramos que es una herramienta que debería de utilizarse una vez que las prácticas con el brazo robótico estén más maduras y permitan una mayor interacción con el alumno.

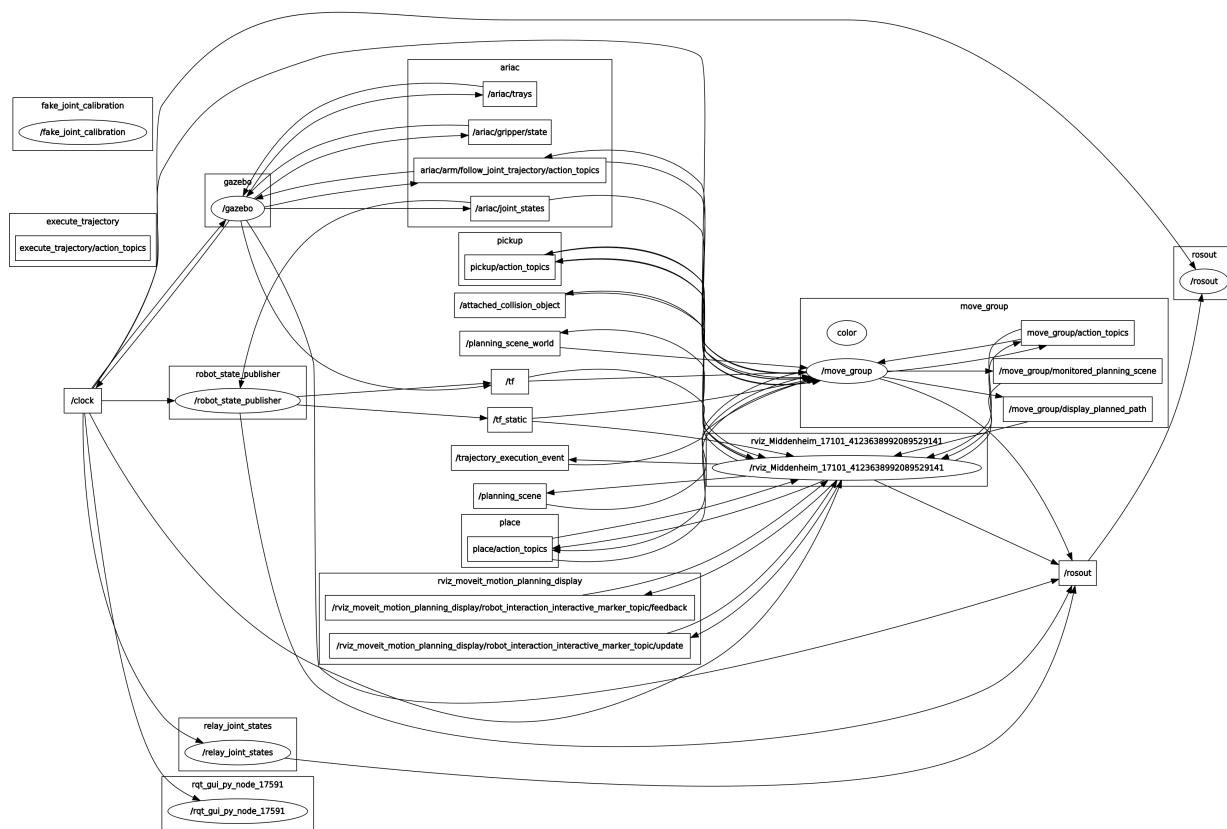


Figura 5.13: Grafo “simplificado” con la relación de nodos y topic de ARIAC, el brazo, MoveIt y rViz.

ROS Message Types	ROS Service Types	ROS Action Types
AllowedCollisionEntry	ApplyPlanningScene	ExecuteTrajectory
AllowedCollisionMatrix	CheckIfRobotStateExistsInWarehouse	MoveGroup
AttachedCollisionObject	DeleteRobotStateFromWarehouse	Pickup
BoundingVolume	ExecuteKnownTrajectory	Place
CollisionObject	GetCartesianPath	
ConstraintEvalResult	GetMotionPlan	
Constraints	GetPlannerParams	
ContactInformation	GetPlanningScene	
CostSource	GetPositionFK	
DisplayRobotState	GetPositionIK	
DisplayTrajectory	GetRobotStateFromWarehouse	
Grasp	GetStateValidity	
GripperTranslation	GraspPlanning	
JointConstraint	ListRobotStatesInWarehouse	
JointLimits	LoadMap	
KinematicSolverInfo	QueryPlannerInterfaces	
LinkPadding	RenameRobotStateInWarehouse	
LinkScale	SaveMap	
MotionPlanDetailedResponse	SaveRobotStateToWarehouse	
MotionPlanRequest	SetPlannerParams	
MotionPlanResponse		
MoveItErrorCodes		
ObjectColor		
OrientationConstraint		
OrientedBoundingBox		
PlaceLocation		
PlannerInterfaceDescription		
PlannerParams		
PlanningOptions		
PlanningScene		
PlanningSceneComponents		
PlanningSceneWorld		
PositionConstraint		
PositionIKRequest		
RobotState		
RobotTrajectory		
TrajectoryConstraints		
VisibilityConstraint		
WorkspaceParameters		

Figura 5.14: Lista de mensajes, servicios y acciones que componen la interfaz de MoveIt.



# Capítulo 6

## Conclusiones

En los capítulos anteriores hemos descrito del problema planteado y las soluciones construidas, así como las elecciones de diseño y los resultados. Para terminar, en este capítulo analizamos de las conclusiones principales y los aportes una vez finalizado el proyecto y las líneas de trabajo que se han abierto con este trabajo.

### 6.1. Aportaciones

Tras observar el trabajo realizado podemos ver que hemos alcanzado los objetivos propuestos en el capítulo 2.1, consiguiendo aportar a las prácticas disponibles para el entorno JdeRobot-Academy mayor variedad y complejidad.

En el caso del primer subobjetivo, se ha creado con éxito un mundo de Gazebo con elevaciones para dar más realismo y atractivo a dos prácticas existentes que manejas coches de Fórmula 1. Para cumplir este objetivo se han creado en total cuatro mundos, uno de ellos plano, otro plano con una línea roja, otro con elevaciones con línea roja y otro con elevaciones sin línea roja. Todos ellos son utilizables y están listos para la realización de prácticas con los coches.

En el caso del segundo subobjetivo hemos encontrado un mundo para Gazebo en el cual se ubica un brazo robótico funcional, y se han creado los componentes software necesarios para desarrollar un teleoperador funcional de ese brazo. Cumplimos, por tanto, la tarea propuesta de construir un teleoperador para un brazo robótico, además de facilitar el desarrollo de nuevas prácticas al proporcionar un mundo donde ya está instalado el brazo.

Además, las soluciones alcanzadas cumplen los requisitos marcados en el capítulo 2.1.1. El desarrollo se ha realizado bajo la arquitectura de JdeRobot en su versión 5.5, Gazebo 7 y

ROS Kinetic. Los componentes software del brazo se han programado utilizando Python, y el resto de componentes tanto del mundo como del modelo 3D en sus respectivos formatos. Los mundos creados han replicado la estructura de los ya existentes y se han incorporado al repositorio oficial de JdeRobot.

Durante la realización de este trabajo se han adquirido una gran cantidad de conocimientos, tanto por necesidad para completar los objetivos como por curiosidad para expandir las posibilidades de desarrollo. El uso del programa de edición 3D Blender ha sido amplio y complejo, pero imprescindible para crear los mundos propuestos. Así mismo, se ha profundizado en las características de Gazebo y sus compatibilidades con archivos de diferentes formatos como Collada, así como los formatos de dichos archivos. También ha sido imprescindible el estudio de JdeRobot y la distribución del código y los ficheros que lo forman para poder realizar aportes significativos a este entorno. Hemos aprendido igualmente mucho sobre ROS y su funcionamiento con Python al realizar el teleoperador del brazo.

## 6.2. Trabajos futuros

Este proyecto sirve de base para futuros trabajos de ampliación y mejora en las prácticas de JdeRobot-Academy. Utilizando los mundos y componentes creados en este trabajo se pueden implementar nuevas funcionalidades en las prácticas o crear nuevos retos para los alumnos. Tres posibles líneas que desarrollar en futuros trabajos se plantean a continuación.

Primero, como complemento a los mundos creados para las prácticas de navegación local y autónoma con el Fórmula 1 se podría mejorar el diseño del modelo del coche.

Segundo, partiendo del teleoperador del brazo robótico se pueden seguir varias líneas de investigación. Una de ellas sería el uso de planificadores de movimiento articulado como MoveIt y la creación de teleoperadores o plugins basados en ellos, de forma que se amplíe la funcionalidad del brazo y las posibilidades académicas que ofrece.

Y tercero, otra línea podría ser el desarrollo de una práctica *pick&place*. Con la incorporación al brazo de algún sensor o cámara se podría añadir al teleoperador una visión desde el brazo. Ampliando las posibilidades del mundo de ARIAC, se podrían generar piezas que el brazo pueda colocar en las bandejas o usar las cámaras y sensores incorporados en el mundo, fuera del brazo. De esta forma se abre un abanico de prácticas en las que explorar las posibilidades del brazo y del teleoperador.

# Bibliografía

- [1] *Libro Blanco de la Robótica: De la investigación al desarrollo tecnológico y aplicaciones futuras.* Editado por CEA - GTRob con subvención del MEC.
- [2] Martinez, A.; Fernández, E. (2013) *Learning ROS for Robotics Programming.* Birmingham, UK: Packt Publishing.
- [3] Joseph, L. (2015) *Mastering ROS for Robotics Programming.* Birmingham, UK: Packt Publishing.
- [4] Página de MediaWiki del proyecto: <http://jderobot.org/Avillamil-tfg>
- [5] Página del repositorio del proyecto: <https://github.com/RoboticsURJC-students/2016-tfg-Alvaro-Villamil>
- [6] Página oficial de Blender: <https://www.blender.org/>
- [7] Página oficial de Gazebo: <http://gazebosim.org/>
- [8] Página oficial de JdeRobot: [http://jderobot.org/Main\\_Page](http://jderobot.org/Main_Page)
- [9] Página oficial de ARIAC: <http://gazebosim.org/ariac>
- [10] Página oficial de documentación de ARIAC: <https://bitbucket.org/osrf/ariac/wiki/Home>
- [11] Página oficial de ROS: <http://www.ros.org/>
- [12] Página oficial de documentación de ROS: <http://wiki.ros.org/>
- [13] Página oficial de MoveIt: <http://moveit.ros.org/>
- [14] Página oficial de rViz: <http://wiki.ros.org/rviz>
- [15] Página oficial de rqt: <http://wiki.ros.org/rqt>

- [16] Página oficial de Qt: <https://www.qt.io/es/>
- [17] Página de texturas: <https://www.textures.com/>
- [18] Página oficial de Wikipedia: <https://www.wikipedia.org/>