

Índice general

1. Introducción	5
1.1. Robótica	5
1.1.1. Historia de la robótica	6
1.2. Educación en robótica	14
1.2.1. Entorno docente con JdeRobot: TeachingRobotics	16
2. Plataforma de Desarrollo	21
2.1. GitHub	21
2.1.1. Git	23
2.2. Blender	24
2.3. Simulador Gazebo	24
2.4. JdeRobot	25
2.5. ROS	26
2.5.1. MoveIt!	27
2.5.2. rviz	28
2.5.3. rqt	28
2.6. ARIAC	28
2.7. Qt	30
3. Práctica Mónaco	31
3.1. Manejo de Blender	31
3.1.1. Interfaz	31

3.2. Circuito plano	35
3.3. Circuito con elevaciones	40
3.4. Mundos para Gazebo	42
4. Brazo robótico	51
4.1. Hablando ROS	53
4.2. Controlando el Brazo	56
Bibliografía	63

Índice de figuras

1.1.	Pato con aparato digestivo de Jacques de Vaucanson.	6
1.2.	Robot de Unimation usado por Ford en 1961.	7
1.3.	Los primeros robots espaciales, el Mars 3 y el Viking I	8
1.4.	Dos ingenieros de la Nasa posan con tres generaciones de Mars Rovers. . . .	9
1.5.	Robot poliarticulado de la empresa Sepro.	11
1.6.	Robot androide.	11
1.7.	Robot androide Nao.	12
1.8.	Diferentes ejemplos de robots zoomórficos	13
1.9.	Robot híbrido usado en la desactivación de bombas.	14
1.10.	Robot ArDrone en Gazebo.	18
1.11.	Robot kobuki en Gazebo.	18
1.12.	Robot Pioneer en Gazebo.	19
2.1.	Escenario de la competición ARIAC.	29
3.1.	Interfaz de Blender.	32
3.2.	Detalle de Manipuladores de transformaciones en 3D.	32
3.3.	Detalle del cursor de Blender.	33
3.4.	Ventana de propiedades.	34
3.5.	Trazado del circuito de Mónaco usado como plantilla	36
3.6.	Detalle de curva Bezier.	36
3.7.	Trazado del circuito de Mónaco superpuesto a la plantilla.	37
3.8.	Segmento del circuito.	38

3.9. Circuito (sólo el trazado).	38
3.10. Circuito plano.	39
3.11. Diferentes vistas del circuito de F1 de Mónaco plano	40
3.12. Fondo para el circuito con elevaciones.	41
3.13. Circuito con elevaciones.	42
3.14. Diferentes vistas del circuito de F1 de Mónaco con elevaciones	43
3.15. Circuitos con una línea roja	44
3.16. Desplegable para exportar ficheros desde Blender.	44
3.17. Opciones de exportación de Collada.	45
3.18. Coches recorriendo el circuito de Mónaco.	49
4.1. Robot PR2.	52
4.2. Robot kuka.	52
4.3. Robot ur10.	53
4.4. Grafo de los <i>nodes</i> y <i>topics</i> de ARIAC.	54
4.5. Grafo de los <i>nodes</i> y <i>topics</i> de ARIAC.	55
4.6. Detalle de las articulaciones del brazo.	58
4.7. Interfaz gráfica del controlador del brazo.	61

Capítulo 1

Introducción

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

1.1. Robótica

Según la Wikipedia[1], *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 intenligencia 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*.

El francés Jacques de Vaucanson creó en el siglo XVIII lo que se consideran los primeros robots de la historia: El flautista, El tamborilero y el Pato con aparato digestivo (*Figura 1.1*). También creó el primer telar completamente automático del mundo. Unos años más tarde, Joseph Jacquard inventa en 1801 una máquina textil programable mediante tarjetas perforadas, y Henri Maillardert construyó una muñeca mecánica que era capaz de hacer dibujos. Al mismo tiempo, en Japón, se creaban juguetes mecánicos que sirven té, disparan flechas y pintan. Podemos considerar esta época el nacimiento de los robots tal como los conocemos actualmente.

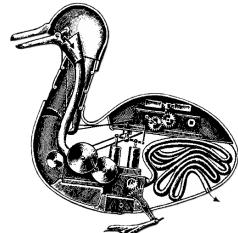


Figura 1.1: Pato con aparato digestivo de Jacques de Vaucanson.

En el año 1921 un escritor checo, Karel Capek, 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* pública “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:

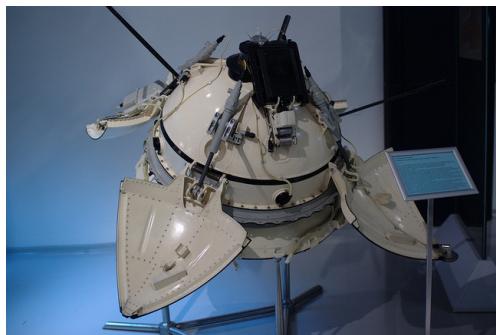
- 1.^a Un robot no hará daño a un ser humano o, por inacción, permitir que un ser humano sufra daño.
- 2.^a Un robot debe hacer o realizar las órdenes dadas por los seres humanos, excepto si estas órdenes entran en conflicto con la 1^a Ley.
- 3.^a Un robot debe proteger su propia existencia en la medida en que esta protección no entre en conflicto con la 1^a o la 2^a 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 evitaran 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. Evidentemente un sistema autónomo hará todo lo posible para no sufrir daños. Como toda tecnología, está diseñada para que funcione y para que mantenga su integridad y su funcionamiento. Su duración será mayor o menor dependiendo de la calidad, pero desde luego no acometerá operaciones destinadas a estropearse. 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 1.2: Robot de Unimation usado por Ford en 1961.

En la década de 1950 se comienzan a desarrollar los primeros robots comerciales, nacidos de las patentes de la década anterior. Debido a la investigación en inteligencia artificial se encontraron maneras de emular el procesamiento de información humana con computadoras electrónicas y se desarrollaron una variedad de mecanismos para probar estas teorías. 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 1.2*), 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



(a) Mars 3



(b) Viking I

Figura 1.3: 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. Carls Sagan posando junto a un modelo del Viking I (b) en el Valle de la Muerte, California

desarrolló en la Standford University. En 1973 la empresa Kuka¹ desarrolla el primer robot industrial con seis ejes electromecánicos, el Famulus. Unos años más tarde, en 1975, la empresa Unimation comercializó un brazo manipulador programable universal llamado PUMA.

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 (*Figura 1.4*), 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 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 1.3 podemos observar ambos robots.

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, tambien conocido como el robot de cuatro sentidos, el famoso Sojourner (*Figura 1.4*) o el Lunar Rover, vehículo con control remoto, y otros mucho mas específicos como el Cypher, un helicóptero robot de uso militar, el guardia de trafico japonés Anzen Taro o los robots mascotas de Sony. En el campo de los robots antropomorfos (androides) se

¹<https://www.kuka.com/>

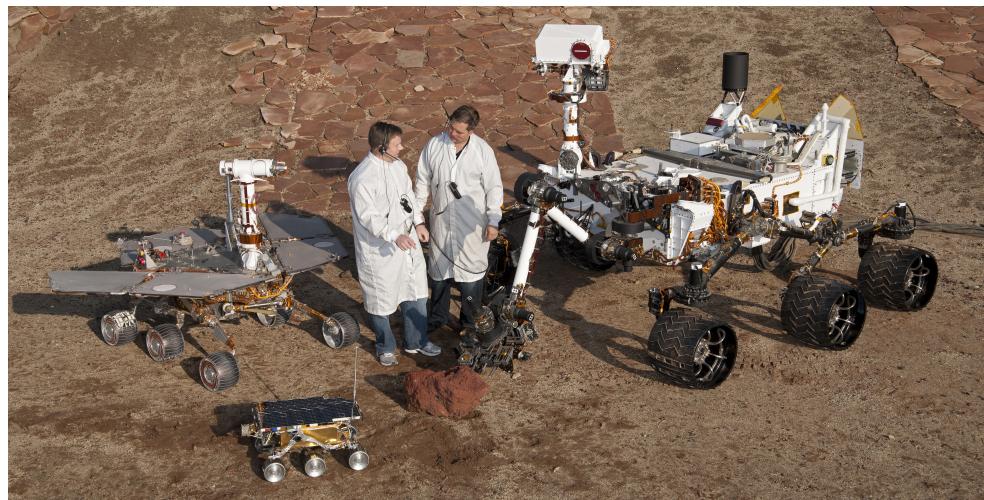


Figura 1.4: Dos ingenieros de la Nasa posan con tres generaciones de Mars Rovers.

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 la Figura 1.4 podemos observar la evolución de los robots usados en la exploración espacial, pues se encuentran tres generaciones de Mars Rover de la NASA. Abajo en el centro nos encontramos con el primer Mars Rover, el Sojourner, que aterrizó en Marte en 1997 como parte del proyecto Mars Pathfinder Project. Es el más pequeño de los tres, con 65 centímetros de largo. A la izquierda se encuentra un vehículo de pruebas MER (Mars Exploration Rover), que mide en torno a los 1.6 metros de largo. Éste es un gemelo operativo del Spirit y del Opportunity que aterrizaron en la superficie marciana en 2004. A la derecha se encuentra un Rover de pruebas, de unos 3 metros de largo, del Laboratorio de Ciencia de Marte (Mars Science Laboratory), responsables del aterrizaje del Curiosity en Marte en 2012.

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

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

2.^a 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.^a 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.^a 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 noventas. 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.

Otra clasificación muy extendida de los robots es según su estructura. La estructura está definida por el tipo de configuración general del Robot. El concepto de metamorfismo, de reciente aparición, se ha introducido para incrementar la flexibilidad funcional de un Robot a través del cambio de su configuración por el propio Robot. El metamorfismo admite diversos niveles, desde los más elementales (cambio de herramienta o de efecto terminal), hasta los más complejos como el cambio o alteración de algunos de sus elementos o subsistemas estructurales. Los dispositivos y mecanismos que pueden agruparse bajo la denominación genérica del Robot, tal como se ha indicado, son muy diversos y es por tanto difícil establecer una clasificación coherente de los mismos que resista un análisis crítico y riguroso. La subdivisión de los Robots, con base en su arquitectura, se hace en los siguientes grupos:

1. Poliarticulados. En este grupo se encuentran los Robots de muy diversa forma y configuración, cuya característica común es la de ser básicamente sedentarios (aunque excepcionalmente pueden ser guiados para efectuar desplazamientos limitados) y estar estructurados para mover sus elementos terminales en un determinado espacio de trabajo según uno o más sistemas de coordenadas, y con un número limitado de grados de libertad. En este grupo, se encuentran los manipuladores, los Robots industriales, los Robots cartesianos y se emplean cuando es preciso abarcar una zona de trabajo relativamente amplia o alargada, actuar sobre objetos con un plano de simetría vertical

o reducir el espacio ocupado en el suelo. Econtramos un ejemplo de robot poliarticulado de la empresa Sepro² en la Figura 1.5.



Figura 1.5: Robot poliarticulado de la empresa Sepro.

2. Móviles. Son Robots con gran capacidad de desplazamiento, basados en carros o plataformas y dotados de un sistema locomotor de tipo rodante. Siguen su camino por telemando o guiándose por la información recibida de su entorno a través de sus sensores. Estos Robots aseguran el transporte de piezas de un punto a otro de una cadena de fabricación. Guiados mediante pistas materializadas a través de la radiación electromagnética de circuitos empotrados en el suelo, o a través de bandas detectadas fotoeléctricamente, pueden incluso llegar a sortear obstáculos y están dotados de un nivel relativamente elevado de inteligencia. Podemos ver un ejemplo de robot móvil en la Figura 1.6.



Figura 1.6: Robot androide.

²<http://www.sepro-group.com/es>

3. Androides. Son los tipos de Robots que intentan reproducir total o parcialmente la forma y el comportamiento cinemático del ser humano. Actualmente, los androides son todavía dispositivos muy poco evolucionados y sin utilidad práctica, y destinados, fundamentalmente, al estudio y experimentación. Uno de los aspectos más complejos de estos Robots, y sobre el que se centra la mayoría de los trabajos, es el de la locomoción bípeda. En este caso, el principal problema es controlar dinámicamente y coordinadamente en el tiempo real el proceso y mantener simultáneamente el equilibrio del Robot. Podemos ver el robot Nao³ en la Figura 1.7, un ejemplo de robot androide de la empresa Aldebaran Robotics, con sede en París.



Figura 1.7: Robot androide Nao.

4. Zoomórficos. Los Robots zoomórficos, que considerados en sentido no restrictivo podrían incluir también a los androides, constituyen una clase caracterizada principalmente por sus sistemas de locomoción que imitan a los diversos seres vivos. A pesar de la disparidad morfológica de sus posibles sistemas de locomoción es conveniente agrupar a los Robots zoomórficos en dos categorías principales: caminadores y no caminadores. El grupo de los Robots zoomórficos no caminadores está muy poco evolucionado. Los experimentos efectuados en Japón basados en segmentos cilíndricos biselados acoplados axialmente entre sí y dotados de un movimiento relativo de rotación. Los Robots zoomórficos caminadores multípedos son muy numerosos y están siendo objeto de experimentos en diversos laboratorios con vistas al desarrollo posterior de verdaderos vehículos terrenos, pilotados o autónomos, capaces de evolucionar en superficies muy accidentadas. Las aplicaciones de estos Robots serán interesantes en el campo de la exploración espacial

³<https://www.aldebaran.com/nao>

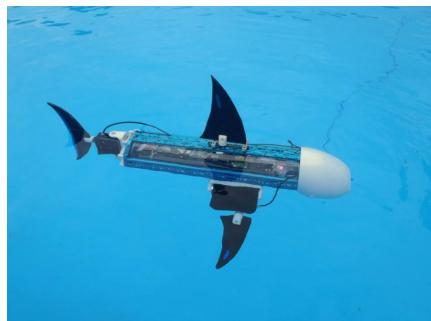
y en el estudio de los volcanes. Podemos ver varios ejemplos de robots zoomórficos en la Figura 1.8.



(a) Robot perro



(b) Robot araña



(c) Robot pez



(d) Robot serpiente

Figura 1.8: Ejemplos de robots zoomórficos: un robot con forma de perro (a), uno con forma de araña (b), uno con forma de pez (c) y uno con forma de serpiente (d)

5. Híbridos. Estos Robots corresponden a aquellos de difícil clasificación, cuya estructura se sitúa en combinación con alguna de las anteriores ya expuestas, bien sea por conjunción o por yuxtaposición. Por ejemplo, un dispositivo segmentado articulado y con ruedas, es al mismo tiempo, uno de los atributos de los Robots móviles y de los Robots zoomórficos. Podemos ver un ejemplo de robot híbrido en la Figura 1.9, en este caso uno de los muchos utilizados para desactivar o explosionar bombas de forma controlada por las fuerzas de seguridad y policía de numerosos países.



Figura 1.9: Robot híbrido usado en la desactivación de bombas.

1.2. Educación en robótica

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 iRobot⁴, la incorporación de aparcamiento automático en coches o incluso asistentes de conducción autónoma como el autopiloto de Tesla⁵ o prototipos de Google o Apple, termostatos inteligentes como el Nest⁶ de Google, o la inmensa variedad de drones del mercado ponen de manifiesto que ésta es una tendencia en auge a nivel global.

Uno de los factores que permiten diseñar, construir y comercializar este tipo de 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. 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.

Este sector 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

⁴<http://www.irobot.es/>

⁵https://www.tesla.com/es_ES/

⁶<https://nest.com/thermostat/meet-nest-thermostat/>

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.

Actualmente en nuestro país, la robótica aparece en los cursos de secundaria, aunque se realiza en la universidad con algunos títulos de grado y postgrado específicos. 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 introducido⁷ la asignatura *Tecnología, programación y robótica* en el currículum oficial de la ESO (Educación Secundaria Obligatoria). En esta etapa se utilizan plataformas como los diferentes robots de LEGO⁸ (Mindstorms, NXT, Evo, WeDo, RCX) o placas con procesadores Arduino 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 o Blockly.

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 XXI⁹:

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

⁷Decreto 48/2015

⁸Lego posee toda una gama pensada para la educación: <https://education.lego.com/>

⁹Según la Asociación para las habilidades del siglo XXI (*Partnership for 21st Century Skills*) <http://www.p21.org/>

- 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.

En la enseñanza superior o universitaria, tradicionalmente se imparten asignaturas de robótica en las escuelas de ingeniería, ya sea industrial, electrónica, informática, etc. En España existe algún grado en robótica y varios másteres. En Estados Unidos, las universidades más punteras en tecnología, como Stanford o el MIT, ya 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 por ejemplo MATLAB¹⁰ y su paquete Simulink¹¹. Otros se centran en el diseño y modelado del robot.

1.2.1. Entorno docente con JdeRobot: TeachingRobotics

Desde la Universidad Rey Juan Carlos se plantea un entorno de enseñanza universitaria llamado *JdeRobot-TeachingRobotics*, dentro de la plataforma JdeRobot (*Sección 2.4*). 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 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 PROGRAMAROBOT¹².

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

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

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

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 resolverlo. 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 dá 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 2.3*), 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*. El mundo de Gazebo no presenta obstáculos para facilitar la programación del robot, y los drones son similares a los AR.Drone de Parrot¹³ (*Figura 1.10*). Hay varios niveles de *ratón* disponibles, cuya dificultad de persecución va en 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.
- *Control visual: sigue líneas.* En esta práctica el alumno debe conseguir que un robot

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

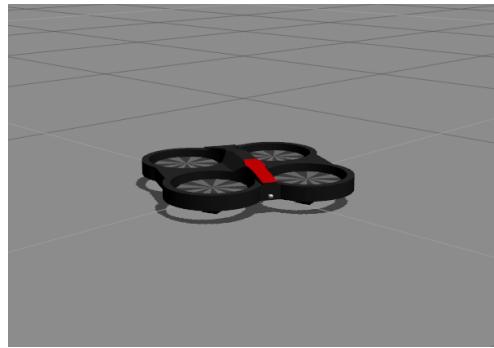


Figura 1.10: Robot ArDrone en Gazebo.

Kobuki (*Figura 1.11*) 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

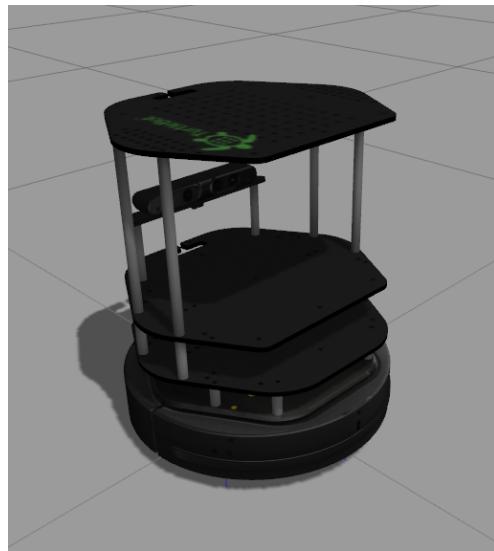


Figura 1.11: Robot kobuki en Gazebo.

- *Fórmula 1: nacegación local.* En esta práctica el alumno debe programar un coche de Fórmula 1 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.
- *TeleTaxi: navegación global.* En esta práctica el alumno debe conseguir que un coche vaya de un punto a otro cualquiera de una ciudad. 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.

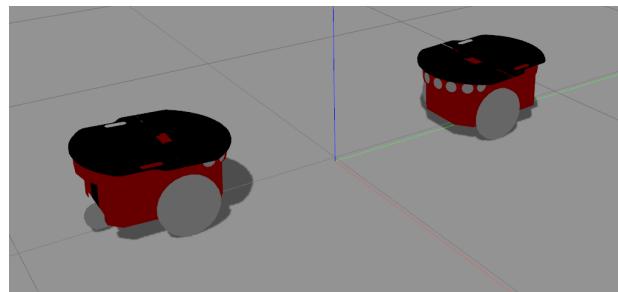


Figura 1.12: Robot Pioneer en Gazebo.

- *Visión: reconstrucción 3D.* En esta práctica el alumno debe conseguir que un robot Pioneer (*Figura 1.12*) 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.

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.

meter cosas de metodología EN EL CAPITULO 2

Capítulo 2

Plataforma de Desarrollo

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.

2.1. GitHub

GitHub[2] es una plataforma de desarrollo colaborativo de software para alojar proyectos utilizando el sistema de control de versiones Git (*Sección 2.1.1*). Utiliza el framework Ruby on Rails por GitHub, Inc. (anteriormente conocida como Logical Awesome). Aloja tu repositorio de código de forma pública (o privada creando una cuenta de pago) y te brinda herramientas muy útiles para el trabajo en equipo dentro de un proyecto. GitHub facilita toda la infraestructura para trabajar en equipos distribuidos a través de una interfaz web. Incluso si no trabajas en equipo, si tienes una copia de tu código

fuente en GitHub, tienes un backup de todo el proyecto completo. Ese backup incluye no sólo el código que tienes ahora sino también de todo el historial de modificaciones que el código ha sufrido desde el primer día. Esta copia la puedes recuperar en cualquier momento y continuar trabajando desde cualquier ordenador.

También permite contribuir a mejorar el software de los demás. Para poder alcanzar esta meta, GitHub provee de funcionalidades para hacer un *fork* y solicitar *pulls*. Realizar un *fork* es simplemente clonar un repositorio ajeno (genera una copia en tu cuenta), y trabajar sobre él para eliminar algún bug, modificar cosas o añadir funcionalidades. Una vez realizadas tus modificaciones puedes enviar un *pull* al dueño del proyecto, el cual podrá analizar los cambios que has realizado fácilmente, y si considera interesante tu contribución, combinarlo con el repositorio original.

En la actualidad, GitHub es mucho más que un servicio de alojamiento de código. Además de éste, se ofrecen varias herramientas útiles para el trabajo en equipo. Entre ellas, caben destacar:

- Una wiki para el mantenimiento de las distintas versiones de las páginas.
- Un sistema de seguimiento de problemas que permiten a los miembros de tu equipo detallar un problema con tu software o una sugerencia que deseen hacer.
- Una herramienta de revisión de código, donde se pueden añadir anotaciones en cualquier punto de un fichero y debatir sobre determinados cambios realizados en un commit específico.
- Un visor de ramas donde se pueden comparar los progresos realizados en las distintas ramas de nuestro repositorio.

Para comenzar a usarlo sólo hay que crearse una cuenta gratuita y ya tendremos acceso a todas sus funcionalidades. Para crear un repositorio solo hay que seleccionar el botón “Create a New Repo”, de la barra de herramientas, habiendo entrado a GitHub con tu cuenta y llenar los datos de nombre y descripción del repositorio.

Para utilizarlo desde el ordenador, GitHub nos ofrece una serie de comandos para introducir en la terminal. Los más usados son:

- `git init`: Con este comando indicamos que vamos a iniciar un proyecto en la carpeta actual, y que lo queremos sincronizar y gestionar mediante GitHub.

- git add: Con este comando añadimos archivos a la siguiente versión o actualización del proyecto. Si hacemos *git add -all* estamos indicando que queremos añadir todo dentro de la carpeta donde estamos: los archivos nuevos, los eliminados, los modificados, etc.
- git commit: Usamos este comando para añadir un mensaje a la versión o actualización o para indicar los cambios hechos. Una forma de usarlo sería *git commit -m "comentario"*
- git pull: Este comando se usa para descargar la última versión del código del repositorio, por ejemplo: *git pull origin master*
- git push: Este comando se usa para subir el código al repositorio, por ejemplo: *git push origin master*
- git remote add origin: Este comando se usa para establecer cuál es el repositorio al que subir el código o del cual descargar la última versión, por ejemplo: *git remote add origin -url del repositorio-*

Para colaborar en un proyecto ajeno simplemente basta con buscarlo dentro de los repositorios en su página web, y luego presionar el botón fork. Esto genera automáticamente una copia del mismo en tu perfil y te permite trabajar sobre esa copia, sin peligro para el autor de que hagas modificaciones en su código. Al terminar tus modificaciones, y hacer los *push* que necesites, podrás presionar Pull Request para enviárselo al autor y que él decida si acepta las modificaciones.

2.1.1. Git

Git[3] es un software de control de versiones diseñado por Linus Torvalds pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando estas tienen un gran número de archivos de código fuente. Se trata de un sistema distribuido de control de código fuente o SCM (*Source Code Management*). Un SCM es una herramienta que nos resuelve una serie de problemas, como saber qué líneas de código han cambiado de una versión a otra, volver a una versión anterior de forma sencilla o saber quién ha cambiado qué partes del código entre otros. En este caso, Git nos aporta herramientas para:

- Auditoría del código: saber quién ha tocado qué y cuándo
- Control sobre cómo ha cambiado nuestro proyecto con el paso del tiempo

- Volver hacia atrás de una forma rápida
- Control de versiones a través de etiquetas: versión 1.0, versión 1.0.1, versión 1.1, etc. Sabremos exactamente que había en cada una de ellas y las diferencias entre cualquiera de ellas dos
- Seguridad: todas las estructuras internas de datos están firmadas con SHA1. No se puede cambiar el código sin que nos enteremos
- Merging y branching extremadamente eficientes
- Mejora nuestra capacidad de trabajar en equipo

2.2. Blender

Blender[4] 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 OpenGL¹ (*Open Graphics Library*) 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)², 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.

2.3. Simulador Gazebo

Gazebo[5] 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

¹<https://www.opengl.org/>

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

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 OSRF³ (*Open Source Robotics Foundation*), la cual también da soporte a ROS, y fué elegido para realizar el DARPA Robotics Challenge⁴ 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

2.4. JdeRobot

JdeRobot[6] 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 la integración con ROS Kinetic. Proporciona un entorno distribuido donde las aplicaciones se forman mediante una colección de componentes asíncronos. Estos componentes utilizan interfaces ICE⁵ 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 com 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.

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*⁶, programa donde Google

³<http://www.osrfoundation.org/>

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

⁵<http://www.zeroc.com/>

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

premia a los estudiantes al completar un proyecto de programación de software libre durante un verano.

2.5. ROS

El Sistema Operativo de Robots, ROS[9] (*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.

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 BSD⁷ 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/subscripción de mensajes 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.

⁷https://es.wikipedia.org/wiki/Licencia_BSD

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.

2.5.1. MoveIt!

MoveIt![11] 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.

2.5.2. rviz

Rviz[12] 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.

2.5.3. rqt

Rqt[13] 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.

2.6. ARIAC

ARIAC[7] (*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 2.1: Escenario de la competición ARIAC.

El objetivo de dicha competición es resolver una serie de problemas dentro del escenario proporcionado. En la Figura 2.1 podemos observar el escenario donde los participantes realizarán las pruebas. Consta de un brazo manipulador ur10 (*Figura 4.3*) 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.

2.7. Qt

Qt[15] 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 3

Práctica Mónaco

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 JdeRobot, en concreto el circuito de Fórmula 1 de Mónaco.

3.1. 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 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.

3.1.1. 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 imagen (*Figura 3.1*). 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.

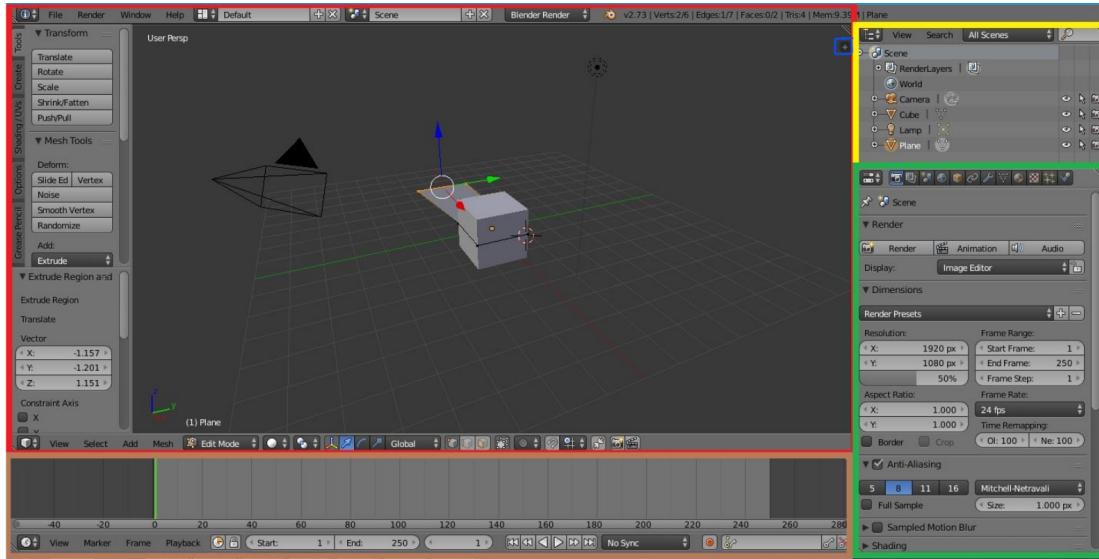


Figura 3.1: Interfaz de Blender.

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:

- Manipuladores de Transformaciones en 3D (*Figura 3.2*): Muestra de manera visual información de las transformaciones a realizar. Cada objeto puede ser transformado de tres maneras: traslación (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 3.2: Detalle de Manipuladores de transformaciones en 3D.

- Cursor 3D (*Figura 3.3*): 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



Figura 3.3: Detalle del cursor de Blender.

hora de trabajar ya que es muy fácil moverlo accidentalmente y al necesitar usarlo se pierde bastante tiempo reubicándolo.

- 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 con 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 3.1 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 se presta 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 segín que condiciones y ver los resultados de aplicar texturas o del renderizado final.

Para poder 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 proporcionara 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.

Ésta ventana posee un botón de propiedades propio marcado en color Azul en la parte superior derecha, el cual despliega la ventana de propiedades (*Figura 3.4*). 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.

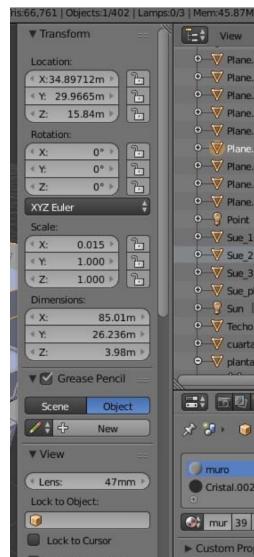


Figura 3.4: Ventana de propiedades.

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 desdoblar elementos entre muchas otras opciones

3.2. 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.

Comenzamos eliminando el cubo que aparece por defecto, desplegando la ventana 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 3.5. 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 queramos 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.

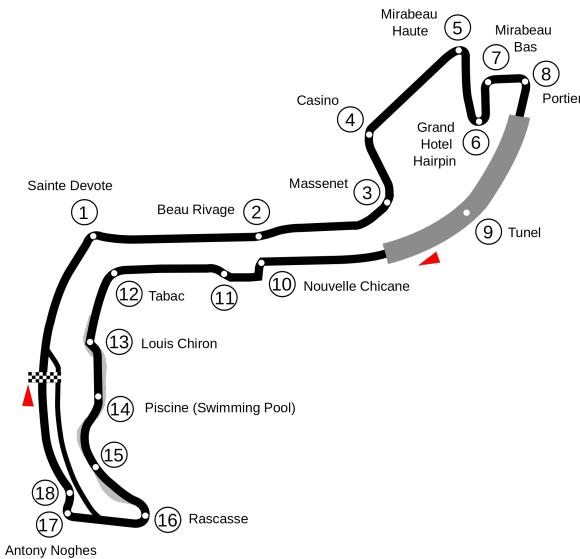


Figura 3.5: Trazado del circuito de Mónaco usado como plantilla

A continuación añadimos una curva “Bezier” (Figura 3.6), 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 3.6, 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.

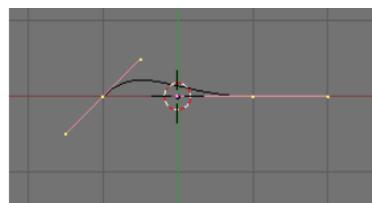


Figura 3.6: Detalle de curva Bezier.

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 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 3.7 (*La línea naranja es la curva Bezier que será el trazado*).

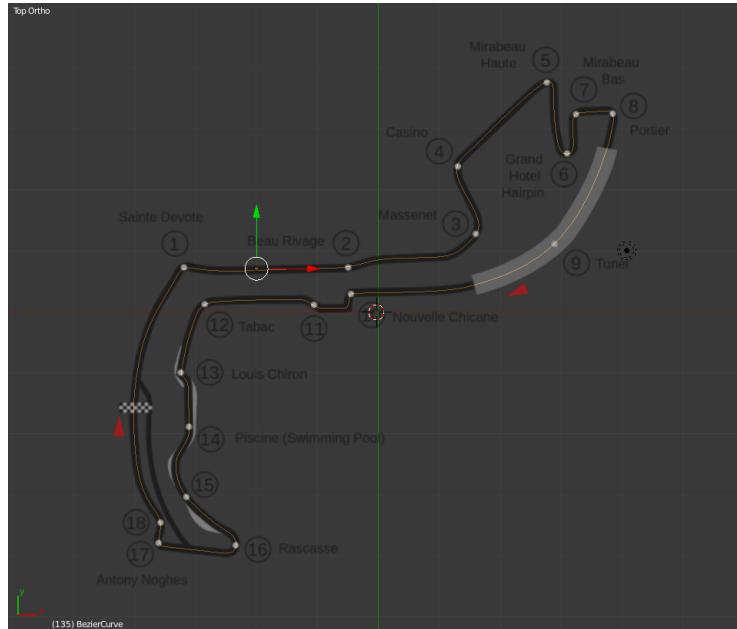


Figura 3.7: Trazado del circuito de Mónaco superpuesto a la plantilla.

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 u 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 textura¹ 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

¹Las imágenes usadas como texturas y materiales en este proyecto han sido obtenidas de manera gratuita de la página textures.com[14]

aparecen en la superficie seleccionada, y multitud de herramientas que hacen mucho más sencillo la edición de texturas. Una vez finalizado el paso de texturizar las caras del objeto obtenemos el segmento de la figura 3.8.

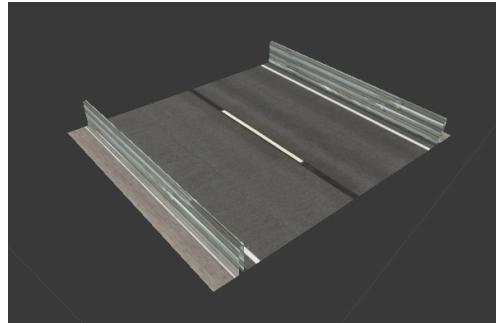


Figura 3.8: Segmento del circuito.

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 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.

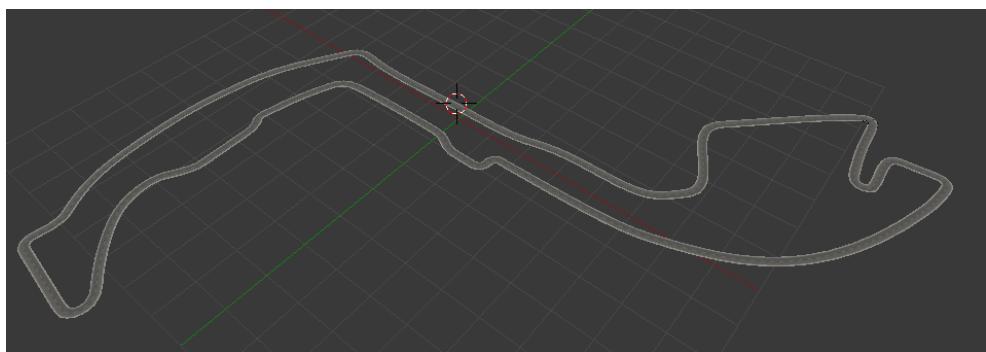


Figura 3.9: Circuito (sólo el trazado).

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 slot se tratase. Despues añadimos



Figura 3.10: Circuito plano.

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 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 3.9.

Una vez obtenido el trazado 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 del circuito. Una vez asignadas todas las texturas, repitiendo los pasos descritos para texturizar el segmento del circuito, obtenemos el circuito de Mónaco (*Figura 3.10*), el cual podemos ver en detalle en la Figura 3.11.

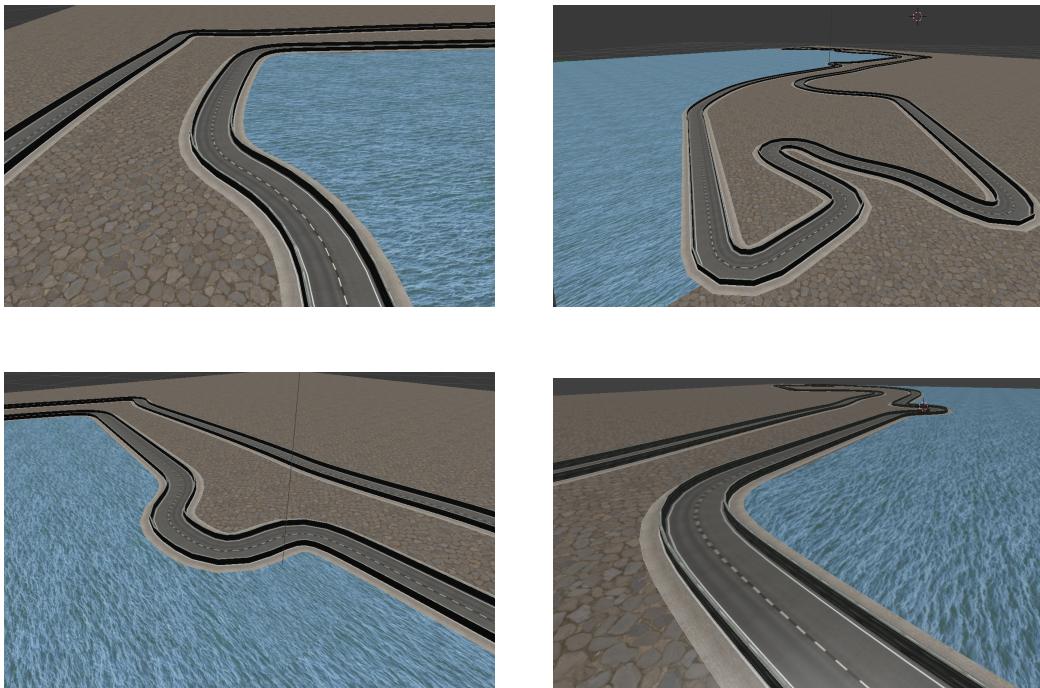


Figura 3.11: Diferentes vistas del circuito de F1 de Mónaco plano

3.3. 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 numero 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 de 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 del circuito real de Mónaco, accedemos a la opción de bloqueo y seleccionamos los ejes x e

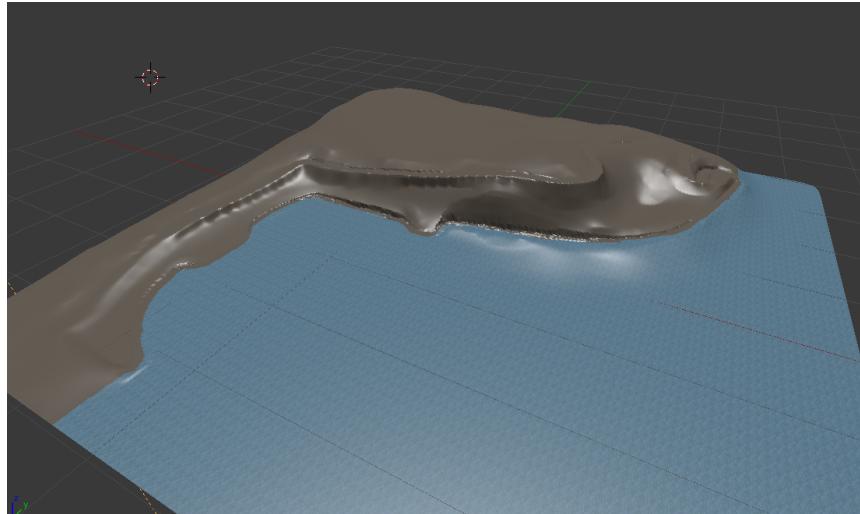


Figura 3.12: Fondo para el circuito con elevaciones.

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 mas 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. Una vez modelado y texturizado el plano, eliminamos el trazado de referencia y obtenemos el suelo sobre el que descansará el circuito (*Figura 3.12*).

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.

Una vez completado este paso podemos dar por finalizado el circuito (*Figura 3.13*). En la Figura 3.14 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 el elevado, con una línea roja en el centro de su recorrido. Dicha línea servirá para realizar prácticas en las que sea necesaria una referencia visual 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 3.15 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 GIMP² 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.

3.4. 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. SDF³ es un formato XML que describe entornos y objetos para simuladores robóticos,

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

³Página de SDF: <http://sdformat.org/>

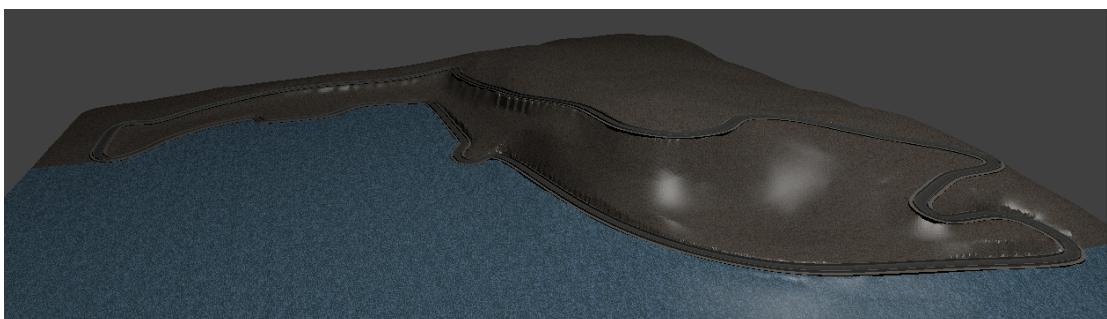


Figura 3.13: Circuito con elevaciones.

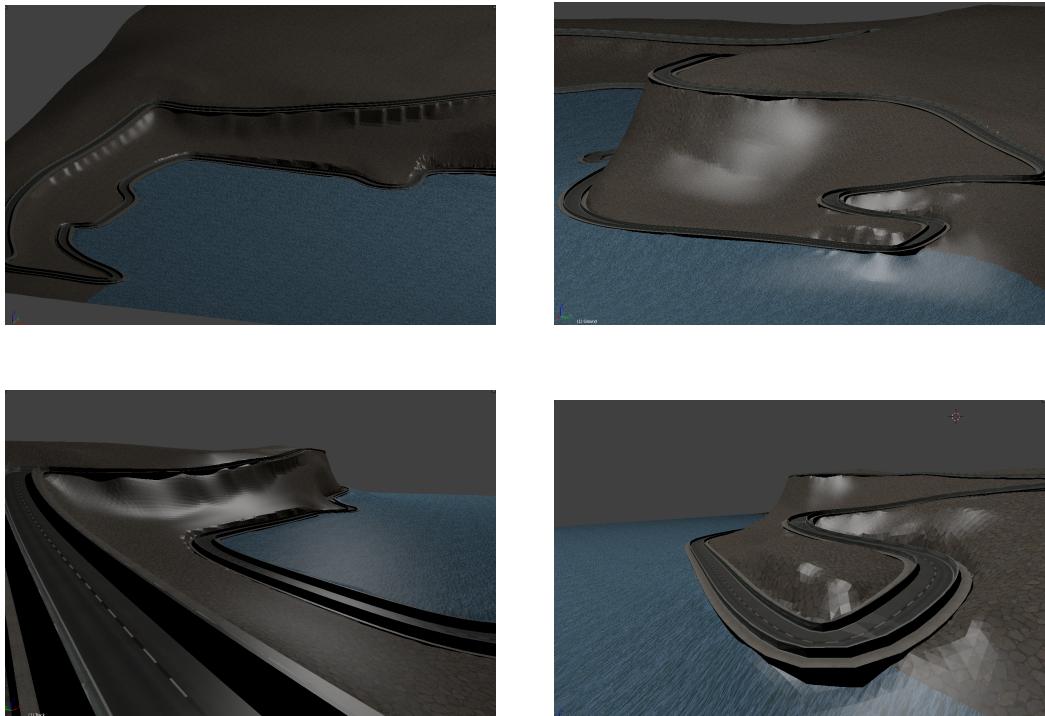


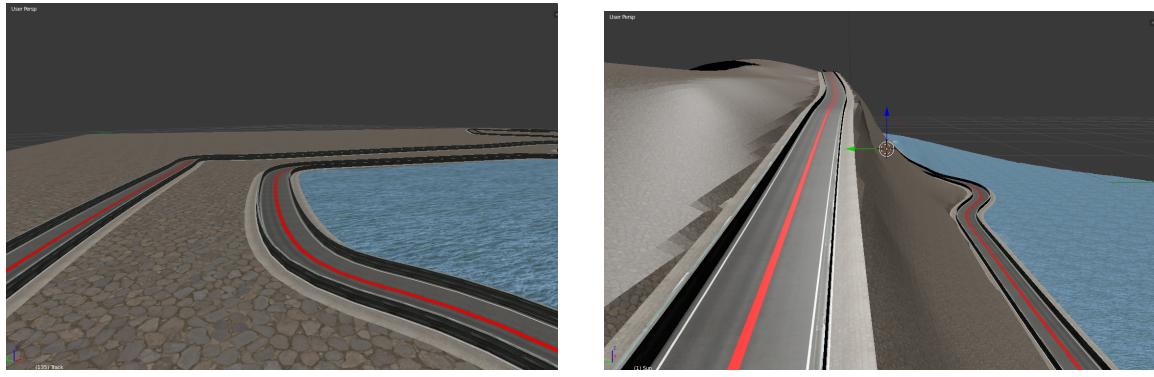
Figura 3.14: Diferentes vistas del circuito de F1 de Mónaco con elevaciones

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 los muchos modelos están en formato .dae⁴. 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 3.16).

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_b y no tenía disponible esta opción. Optamos por instalar desde fuente la versión 2.78_c que sí lo tiene.

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 3.17. Si no se activan las casillas adecuadas

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



(a) Mónaco plano

(b) Mónaco con elevaciones

Figura 3.15: Circuitos de mónaco editados para contener una línea roja en su trazado, tanto el plano (a) como el que contiene elevaciones (b)

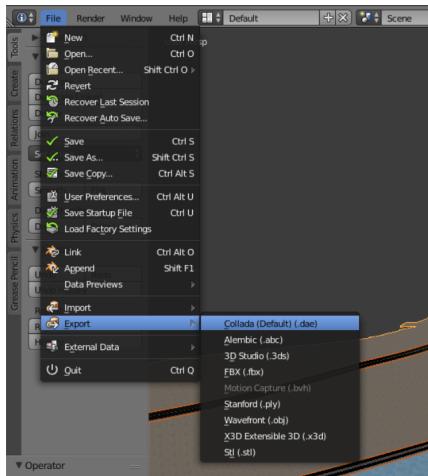


Figura 3.16: Desplegable para exportar ficheros desde Blender.

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 3.8). 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*

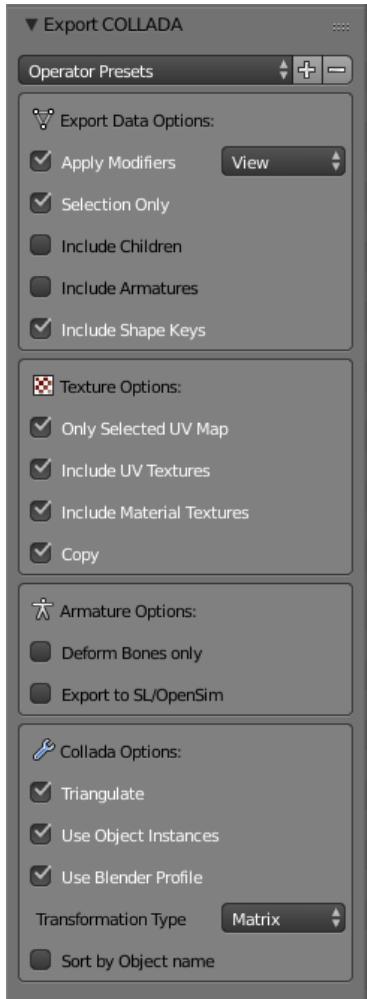


Figura 3.17: Opciones de exportación de Collada.

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 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</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 mínimamente 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 vé 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.

A continuación creamos 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 este:

```

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. 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 3.18: 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 JdeRobot⁵ en GihHub. De esta forma logramos incorporar nuestros progresos al código oficial de JdeRobot, y que cualquiera que se descargue ésta plataforma pueda disfrutar de los mundos que hemos

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

creado. En la figura 3.18 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 4

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. 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: De la empresa Clear Path Robotics¹, responsables también del kobuki o turtlebot. Se trata de un robot con apariencia de androide (*Figura 4.1*). 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 ROS² observamos que no da soporte más allá de Indigo, lo cual puede suponer un problema.
- kuka: La empresa Kuka³ 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 4.2, aunque la plataforma donde se apoya el brazo puede ser diferente o no estar. En su página de ROS⁴ vemos que soporta

¹<https://www.clearpathrobotics.com/>

²<http://wiki.ros.org/Robots/PR2>

³<https://www.kuka.com/>

⁴<http://wiki.ros.org/kuka>



Figura 4.1: Robot PR2.

las versiones de Indigo y Kinetic, pero mediante la instalación del paquete externo ROS-Industrial⁵.



Figura 4.2: Robot kuka.

- ur10: De la empresa Universal Robots⁶. Es un brazo simple (*Figura 4.3*), al igual que sus hermanos el ur3 y el ur5. La diferencia entre estos no va más allá del tamaño y fuerza del robot. En su página de ROS⁷ observamos que no da soporte más allá de Indigo, lo cual puede suponer un problema.

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

⁵<http://wiki.ros.org/Industrial>

⁶<https://www.universal-robots.com/es/>

⁷http://wiki.ros.org/ur_gazebo



Figura 4.3: Robot ur10.

de intentar solucionar los problemas por nuestra cuenta, vemos que ninguno de ellos llega a funcionar de forma correcta bajo nuestros requisitos.

Recurrimos a los escenarios de ARIAC (*Sección 2.6*) para desarrollar nuestro trabajo. Como podemos ver en la Figura 2.1 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. Es de gran ayuda la página de documentación de ARIAC[8] donde se detallan las interfaces de comunicación de los elementos del escenario.

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

4.1. Hablando ROS

Para aprender el funcionamiento de ROS vamos a la documentación oficial[10]. Una vez allí nos damos cuenta de que 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 grafos. 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 mucho mas 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.

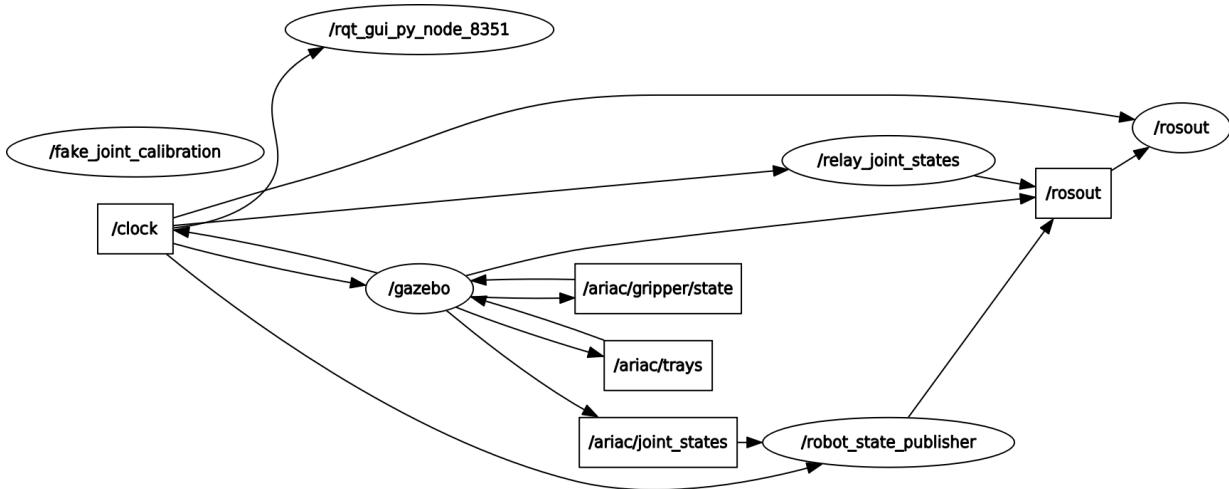


Figura 4.4: Grafo de los nodes y topics de ARIAC.

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*, por ejemplo un sensor de temperatura o de proximidad, o recibir información del topic haciendo *subscribe*, para tomar decisiones o procesar los datos recibidos. 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 conseguir 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*, *booleanos*, 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.

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 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 4.4. 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 es subscripto 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*.

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.

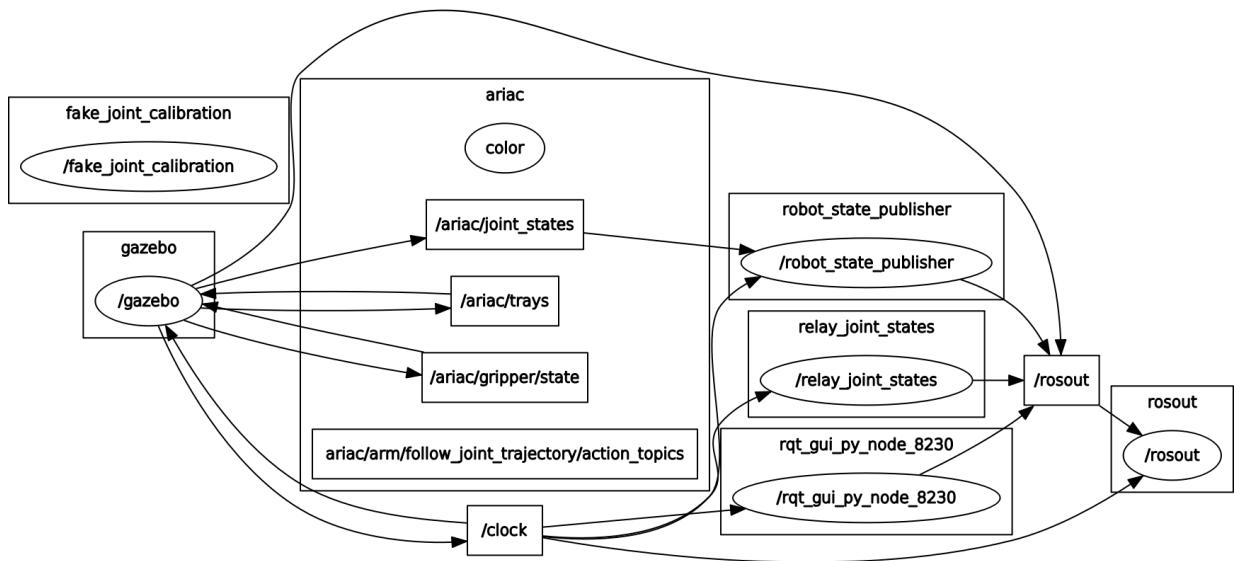


Figura 4.5: Grafo de los nodes y topics de ARIAC.

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 ariac: *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 este está el nodo de *rqt* que nos permite obtener esta imagen. A la 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.

4.2. Controlando el Brazo

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 recibirllos, por lo que no nos preocupamos por el.

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 : Esta es la articulación del codo, la del medio del brazo.
- linear_arm_actuator_joint : Esta 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 : Esta es la articulación del hombro, es decir, la más cercana a su base. En concreto esta es la articulación que controla la elevación del brazo. Realiza giros sobre un imaginario eje y.
- shoulder_pan_joint : Esta es la articulación del hombro, es decir, la más cercana a su base. En concreto esta es la articulación que controla la orientación del brazo y nos permite girarlo. Realiza giros sobre un imaginario eje z.
- wrist_1_joint : Esta es la articulación de la muñeca, la más alejada de la base. En concreto esta articulación controla la elevación de la muñeca y nos permite subirla y bajarla. Realiza giros sobre un imaginario eje y.
- wrist_2_joint : Esta es la articulación de la muñeca, la más alejada de la base. En concreto esta articulación controla el giro de la muñeca y nos permite rotarla verticalmente. Realiza giros sobre un imaginario eje z.
- wrist_3_joint : Esta es la articulación de la muñeca, la más alejada de la base. En concreto esta articulación controla el giro de la mano y nos permite rotar la muñeca horizontalmente. Realiza giros sobre un imaginario eje x.

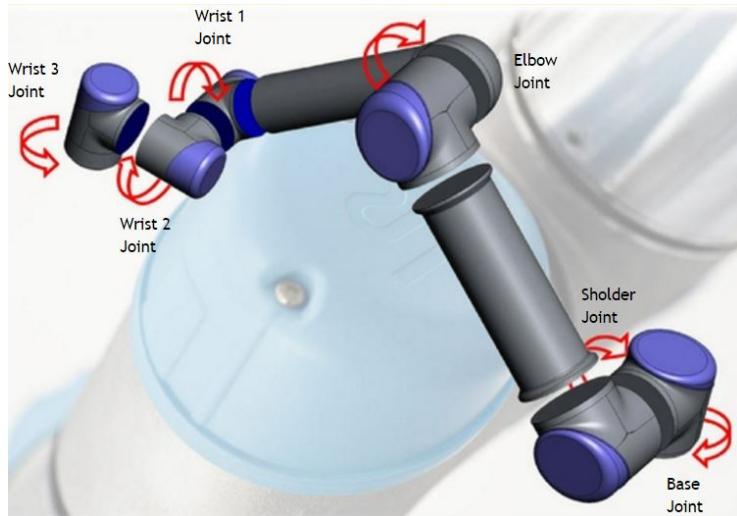


Figura 4.6: Detalle de las articulaciones del brazo.

- `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 4.6 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 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 nomvres 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 com 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 anterios, 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.

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. De esta manera aprendemos a obtener información para conocer la posición inicial del brazo. A continuación hacemos

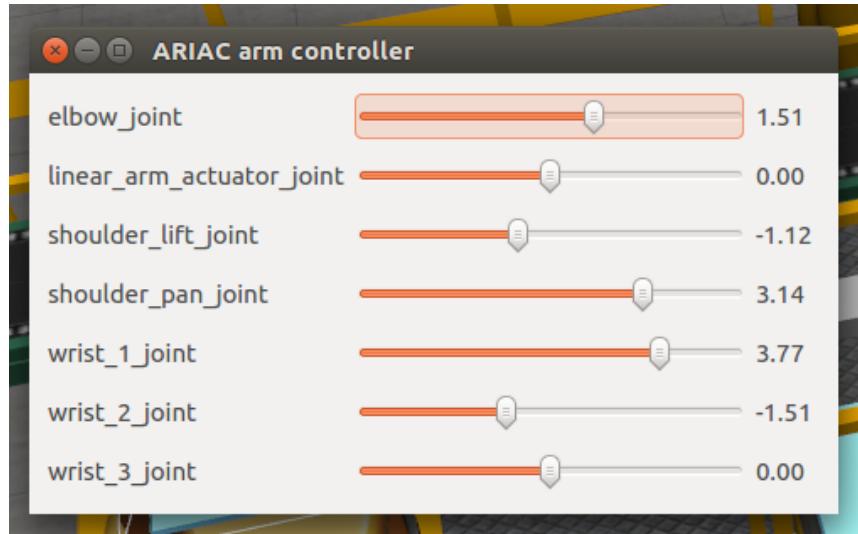


Figura 4.7: Interfaz gráfica del controlador del brazo.

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. Nos servimos de Qt para desarrollar la interfaz visual. Crearemos unos deslizadores para cambiar los valores de posición de cada articulación por separado, consiguiendo de una forma intuitiva saber donde están los límites de cada articulación y cuál es su posición respecto al total de su movimiento. También crearemos unas etiquetas que muestren 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 dà 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.
- *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.
- *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.
- *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.
- *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.

En la Figura 4.7 podemos ver la apariencia final del controlador del brazo.

Bibliografía

- [1] Página oficial de Wikipedia: <https://www.wikipedia.org/>
- [2] Página oficial de GitHub: <https://github.com/>
- [3] Página oficial de Git: <https://git-scm.com/>
- [4] Página oficial de Blender: <https://www.blender.org/>
- [5] Página oficial de Gazebo: <http://gazebosim.org/>
- [6] Página oficial de JdeRobot: http://jderobot.org/Main_Page
- [7] Página oficial de ARIAC: <http://gazebosim.org/ariac>
- [8] Página oficial de documentación de ARIAC: <https://bitbucket.org/osrf/ariac/wiki/Home>
- [9] Página oficial de ROS: <http://www.ros.org/>
- [10] Página oficial de documentación de ROS: <http://wiki.ros.org/>
- [11] Página oficial de MoveIt: <http://moveit.ros.org/>
- [12] Página oficial de rViz: <http://wiki.ros.org/rviz>
- [13] Página oficial de rqt: <http://wiki.ros.org/rqt>
- [14] Página de texturas: <https://www.textures.com/>
- [15] Página oficial de Qt: <https://www.qt.io/es/>