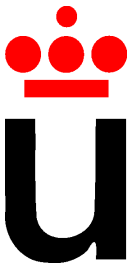


UNIVERSIDAD REY JUAN CARLOS

INGENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS

Escuela Superior de Ciencias Experimentales y Tecnología

Curso académico 2006-2007

Proyecto Fin de Carrera

Jdeneo.c - Una plataforma de desarrollo de aplicaciones robóticas.

Tutor: José María Cañas Plaza

Autor: Jesús Ruiz-Ayúcar Vázquez

"In the fifties, it was predicted that in 5 years robots would be everywhere.
In the sixties, it was predicted that in 10 years robots would be everywhere.
In the seventies, it was predicted that in 20 years robots would be everwhere.
In the eighties, it was predicted that in 40 years robots would be everywhere."
Marvin Minsky

Este documento forma parte del Proyecto Fin de Carrera de la titulación de Ingeniería Técnica en Informática de Sistemas de la Universidad Rey Juan Carlos de Madrid realizado por Jesús Ruiz-Ayúcar Vázquez

El desarrollo de jdeneo.c forma parte del Grupo de Robótica de la Universidad Rey Juan Carlos (www.robotica-urjc.es).

Agradecimientos.

En primer lugar quisiera agradecer toda la compañía y ayuda recibida por el Grupo de Robótica de la Universidad Rey Juan Carlos, especialmente a Pablo Barrera y a Antonio Pineda.

Gracias a mis padres, por haberme motivado a realizar mis estudios y por haber hecho todo lo posible por que así fuera. Y a Belén por animarme en todo momento.

Quiero agradecer también la ayuda recibida en el canal `#gtk+` de la red irc `freenode.org`.

Por último, mis más sinceros agradecimientos a José María Cañas por todo el esfuerzo que ha puesto, por su buen asesoramiento y por saber contagiarme su entusiasmo. En definitiva, por ser todo lo que se le puede pedir a un mentor.

Resumen.

El comportamiento de un robot viene directamente determinado por la programación realizada sobre el mismo. Con la creciente complejidad de estos comportamientos se hace necesaria una correcta estructuración del *software*, que asegure, en la medida de lo posible, un diseño eficiente y escalable. Para ello surgen las plataformas de desarrollo para aplicaciones robóticas, que ofrecen acceso de alto nivel a los recursos del robot y utilidades para asistir en la programación de comportamientos. Una de ellas es JDE, y su implementación en código C es `jde.c`.

El objetivo de este proyecto es crear `jdeneo.c`, una nueva implementación de la arquitectura teórica JDE. Empleando como punto de partida `jde.c`, hemos desarrollado esta nueva infraestructura solventando las mayores deficiencias de su predecesor, apoyándonos en todo momento en bibliotecas robustas y modernas. Se ha logrado cumplir todos los objetivos fijados inicialmente, creando, como han demostrado los ejemplos de uso, una infraestructura ágil y sólida.

Una de las principales características de `jdeneo.c` es la posibilidad de extender su funcionalidad mediante *plugins*, permitiendo de ese modo agregar nuevos comportamientos en tiempo de ejecución. Para realizarlo hemos empleado la biblioteca de propósito general GLib. También se ha añadido visualización tridimensional de la escena alrededor del robot de forma nativa, gracias a la biblioteca OpenGL. Además, se ha diseñado una interfaz gráfica de usuario empleando GTK+, una biblioteca gráfica potente empleada en multitud de proyectos.

Para comprobar el correcto funcionamiento de la infraestructura programada, se han realizado pruebas de todos los mecanismos, incluyendo un ejemplo real en el que genera un comportamiento de navegación segura basado en la técnica de navegación local *Campo de Fuerzas Virtuales* (VFF).

Índice general

Capítulo 1

Introducción

El 25 de Enero de 1921 el escritor checo Karel Čapek estrenó su obra "R.U.R." en el Teatro Nacional de Praga. Fue en esa obra en la que se dio a conocer la palabra *robot* para describir una *persona artificial*. Dos años después, en 1923, la obra se presentó en el idioma Inglés, propagándose desde entonces el término *robot* al resto de idiomas. ¿Pero qué entendemos realmente por robot?

1.1. Introducción a la robótica

Según la wikipedia[Wik, 2006] un robot "*es un dispositivo o grupo de dispositivos electro-mecánicos o bio-mecánicos que pueden generar comportamientos autónomos o preprogramados*". No obstante, existen otras acepciones para el término "robot", como son los robots web[1] que navegan por la estructura de la *world wide web* en busca de documentos. Sin embargo, la definición más común se refiere a aquellos dispositivos físicos capaces de interactuar con su entorno.

La idea de seres artificiales data, al menos, de la antigua leyenda de *Cadmus*, en la que los dientes de un dragón se convirtieron en soldados, o el mito de Pigmalión, cuya estatua Galatea cobró vida. Podemos encontrar conceptos más similares a lo que hoy día entendemos como robot desde el año 450 a.C., cuando el matemático griego Archytas postuló un pájaro mecánico. Más tarde, entre los años 10 y 70 d.C. Herón de Alejandría realizó numerosas innovaciones en el campo de los autómatas, incluyendo uno que podía hablar.

Uno de los primeros diseños registrados de robots humanoides fue realizado por Leonardo da Vinci alrededor de 1495. Las notas de da Vinci contenían dibujos detallados de un guerrero capaz de sentarse, mover sus brazos y cabeza, y sonreír. Sin embargo, muchos consideran que el primer robot, en el sentido de un dispositivo teleoperable, fue ideado por el físico Nicola Tesla, exhibido en 1898 en el Madison Square Garden. No fue hasta 1948 cuando se creó el primer robot autónomo por W. Grey Walter, de la Universidad de Bristol, Inglaterra.

Los desarrollos actuales de robótica persiguen un gran número de objetivos. Encontramos por ejemplo la búsqueda por lograr modos de caminar similares a los de humanos o animales, como es el caso del robot ASIMO (Figura 1.1(a)). Este área de investigación es conocida como *robótica biomórfica*. Otro de los campos en los que se investiga es en el desarrollo de máquinas

[1] Google, por ejemplo, utiliza el conocido `googlebot` para rastrear el contenido de la web y poder indexarlos.

que puedan realizar tareas desagradables o peligrosas. O los vehículos guiados de forma automática, como los robots móviles para el transporte de materiales. En los últimos años también han comenzado a instalarse en hogares, como el robot aspirador Roomba[2] (Figura 1.1(b)), por lo que surgen nuevas líneas de investigación.

(a) ASIMO, de la casa Honda.

(b) Robot aspiradora Roomba.

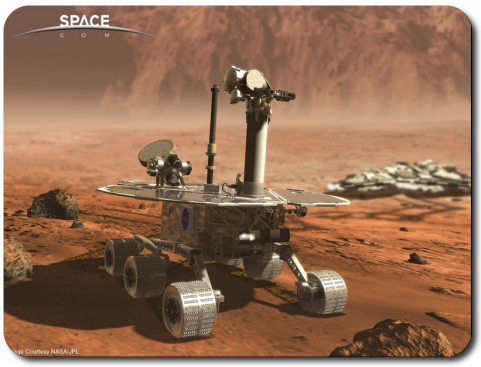
Figura 1.1: Líneas de investigación actuales.

Una de las áreas en las que la robótica se ha implantado fuertemente es en la medicina. Gracias a ellos es posible realizar tareas muy delicadas de manera exitosa, o permitir que un cirujano realice una operación manejando el robot de manera remota.

Las exploraciones espaciales suponen otro campo de aplicación robótica, permitiendo una amplia variedad de funciones, como por ejemplo pruebas de vuelo planetarias, brazos mecánicos para manipulación de elementos, pruebas atmosféricas o robots terrestres. Uno de los más conocidos es el robot `Pathfinder`: Teleoperado desde La Tierra, permitió analizar rocas e imágenes de la superficie de Marte.

Sin embargo, donde quizás sea más notable el impacto de la robótica sea en líneas de producción industriales, especialmente en cadenas de montajes de vehículos. Se utilizan para muchas funciones, tales como fabricación, ensamblaje y soldadura de piezas, paletizado y logística de almacén. Estos robots son definidos como manipuladores programables en tres o más ejes multipropósito, controlado automáticamente y reprogramables.

[2] http://irobot.com/



(a) Representación del robot `Pathfinder`

(b) Cadena de robots soldadores.

Figura 1.2: Desarrollos actuales.

La robótica es un área de conocimiento multidisciplinar donde confluyen muchos campos, como la inteligencia artificial, la visión computacional, la mecánica o la psicología [Plaza, 2004]. Una de las ramas más interesantes quizás sea la robótica móvil, cuyo objetivo es lograr que las máquinas realicen de forma autónoma tareas con la flexibilidad y robustez que muestra el ser humano al realizarlas.

Más allá de su forma y apariencia física, los robots se componen de tres elementos principales[Maya, 2006]:

1. Sensores. Constituyen el sistema de percepción del robot. Son dispositivos físicos que miden cantidades físicas, como por ejemplo distancias a obstáculos, inclinación, sonido, etc. En general, son inexactos y muy sensibles al ruido.

2. Actuadores. Son los dispositivos encargados de generar movimiento en los elementos del robot. Lo que hace un actuador es parte de lo que va a poder hacer el robot. Hay de varios tipos, siendo los más destacables los hidráulicos, los neumáticos y los motores eléctricos.

3. Controlador y ordenador. Son los dispositivos *hardware* que coordinan la actuación y la percepción del robot. Dicha coordinación viene directamente determinada por la programación que se realice sobre el *hardware* de cómputo.

1.2. Programación de robots

La programación de robots supone un esfuerzo para los desarrolladores debido a que son dispositivos muy heterogéneos – Existe un gran número de dispositivos *hardware* disponibles en la robótica[Plaza *et al.*, 2007]. Podemos, por ejemplo, encontrar modelos cuyos actuadores son patas y un cuello mecánico, y cuyos sensores son un par estéreo de cámaras. O bien robots movidos por ruedas y con una unidad láser como sensor.

Además hay ciertos requisitos que deben cumplir las aplicaciones que generan comportamientos en los robots. Uno de ellos es el tiempo de reacción, el cual se debe aproximar lo más

posible a tiempo real, exigiendo programas ágiles que garanticen vivacidad de respuesta. De igual modo, se hacen necesarios mecanismos de concurrencia, para garantizar que un conjunto de tareas se ejecuten en todo momento.

Todos estos motivos hacen de la programación robótica una disciplina muy particular dentro de la informática. Antiguamente la implementación de este tipo de programas era *artesanal*, en el sentido de que no existía ninguna metodología más o menos estandarizada para garantizar una arquitectura *software* coherente. Únicamente se utilizaban los *drivers* que proporcionaba el fabricante de cada robot para realizar estas aplicaciones. La organización de la aplicación tampoco seguía ningún método concreto, por lo que surgía también el problema de cómo abordar el diseño: Una mala organización de la aplicación implicaba que, a medida que el grado de complejidad aumentaba, el desarrollo no fuera sostenible. Con el tiempo este tipo de desarrollo se ha vuelto obsoleto, y ha sido sustituido por los entornos de programación de robots.

Para solventar todas las necesidades implícitas a la programación de robots surgieron los entornos de programación para aplicaciones robóticas [Plaza, 2004]. Estas arquitecturas *software* ofrecen al programador todos los recursos del robot de forma uniforme, ocultando de ese modo la heterogeneidad que ya hemos comentado, y tratan también de garantizar un sistema robusto y fiable, facilitando bibliotecas que simplifiquen tareas comunes en la programación de aplicaciones robóticas. En muchos casos estas arquitecturas se implementan como un *middleware*.

En los últimos años se han creado varias plataformas y *middleware* para ayudar en ese desarrollo. Los principales fabricantes proporcionan a los programadores sus propios kits de desarrollo. Tal es el caso de ActivMedia, que proporciona la arquitectura ARIA[3] para sus robots Pioneer (Ver figura 1.3(b)), Open-R[4], la plataforma de desarrollo que Sony facilita a los desarrolladores de aplicaciones del perro Aibo (Ver figura 1.3(a)), o ERSP, desarrollada por Evolution Robotics[5].

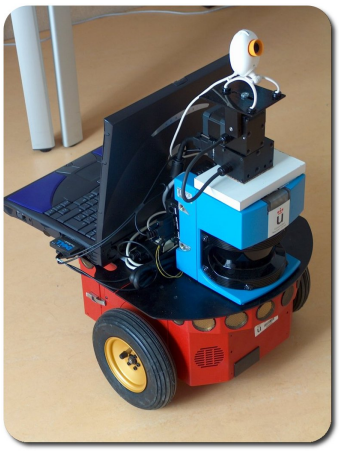
Cada plataforma encapsula la funcionalidad de diferentes maneras, dando diferentes niveles de abstracción y haciendo más fácil la generación de comportamientos en robots. Las más modernas proporcionan métodos para reusar código, con el consiguiente incremento de productividad, imponen restricciones en la organización del software, y dividen la funcionalidad en pequeños bloques.

1.3. Arquitectura *software* basada en JDE

El problema de cómo generar comportamientos autónomos en robots es un problema fundamental en la robótica. A lo largo de los años han surgido varias escuelas que proponen diferentes paradigmas de programación o arquitecturas cognitivas. Estas bases cognitivas son propuestas

[3]http://www.activrobots.com/SOFTWARE/aria.html
[4]http://openr.aibo.com/
[5]http://www.evolution.com/

(a) Perrito Aibo de la casa Sony



(b) Robot Pioneer

Figura 1.3: Robots de propósito general.

de metodologías para afrontar la generación de comportamientos autónomos para robots y escalar la complejidad. Definen de forma teórica una infraestructura, que en la práctica se plasman en arquitecturas software.

No existe una solución universal para todos los entornos y comportamientos. Tradicionalmente encontramos las siguientes bases cognitivas:

1. **Sistemas deliberativos**. Estos sistemas están muy relacionados con la inteligencia artificial, en tanto que la secuencia de acciones a realizar para lograr un objetivo se planifica anticipadamente. Esta planificación se realiza para calcular el mejor camino hasta el objetivo.

2. **Sistemas reactivos**. Esta corriente, a diferencia de los sistemas deliberativos, no utiliza capacidad de predicción. Su funcionamiento se basa en realizar cálculos dinámicos basándose en los sensores, para tomar decisiones "al vuelo".

3. **Sistemas de control híbridos**. La principal característica de este tipo de sistemas es la combinación de los sistemas deliberativos y de los sistemas reactivos. De este modo se auna la capacidad de anticipación de los sistemas deliberativos con la flexibilidad de los sistemas reactivos.

4. **Sistemas basados en comportamientos**. Apuestan por un control distribuido, de modo que los diseños se descomponen en comportamientos con objetivos concretos, que combinados entre sí interactúan para lograr un mismo propósito. JDE, la arquitectura cognitiva que usaremos a lo largo de este proyecto, pertenece a este tipo de sistemas.

Algunas de las infraestructuras existentes, no siguen ninguna arquitectura cognitiva. Tal es el caso del entorno `player/stage`[Gerkey *et al.*, 2000][Vaughan, 2000].

Una de las especificaciones teóricas que estudia cómo debe comportarse una arquitectura es JDE (Jerarquía Dinámica de Esquemas). Surge como una tesis doctoral dentro del Grupo de Robótica de la Universidad Rey Juan Carlos[Plaza, 2003]. Pretende definir una aproximación a los sistemas basados en comportamientos que facilite la programación en robótica móvil.

Podemos dividir la arquitectura JDE en tres puntos principales. Por un lado, se han separado dos aspectos distintos: la percepción y el control. Por otro lado, las unidades mínimas que definen comportamientos son los esquemas. Finalmente, todos los esquemas pueden combinarse dinámicamente en jerarquías.

Percepción y control

Vemos, por tanto, que el problema del comportamiento autónomo se divide en dos partes: la percepción y la actuación. Para lograr un comportamiento completamente autónomo la arquitectura debe contemplar ambas y sus interrelaciones. La percepción se encarga de extraer la información relevante para los objetivos del robot, mientras que el control se encarga de calcular la respuesta adecuada para lograr los objetivos.

Esquemas

La arquitectura se divide en pequeñas unidades de comportamiento llamadas esquemas. Se definen como *un flujo de ejecución independiente con un objetivo*. Esta ejecución se realiza de forma iterativa, es decir, funcionan cíclicamente. Además, se pueden activar y detener a voluntad en cualquier momento.

Como la arquitectura distingue entre tareas de actuación y de percepción, tendremos esquemas perceptivos y esquemas de actuación respectivamente. Los perceptivos elaboran representación, y los de actuación tomas decisiones de control. De este modo, los esquemas perceptivos siempre van acompañados de los de actuación, y son activados por ellos. Esta distinción favorece la reutilización de los esquemas para diferentes comportamientos.

Combinación en jerarquía

Atendiendo al comportamiento deseado el conjunto de esquemas se organizará en capas, creando en última instancia una jerarquía. Esta jerarquía se construye cuando la salida de un esquema de control son activaciones de otros esquemas de nivel inferior, o cuando un esquema perceptivo se apoya en otros esquemas perceptivos más sencillos activándolos para elaborar su propio estímulo. La jerarquía permite elevar el grado de abstracción y aumentar de manera exitosa la complejidad del comportamiento deseado.

1.3.1. jde.c

Una de las implementaciones de la arquitectura cognitiva JDE es `jde.c`. Inicialmente fue desarrollado en la tesis doctoral que definió JDE[Plaza, 2003], actualmente es fruto del trabajo

del grupo de robótica de la Universidad Rey Juan Carlos, la cual hace de `jde.c` un proyecto vivo en continua evolución empleado para la investigación en generación de comportamientos y como herramienta docente en asignaturas de robótica. Además, ha sido el precursor de este proyecto fin de carrera. Además de `jde.c`, existe otra implementación de la arquitectura JDE: `jde+`, desarrollado íntegramente en el lenguaje `C++`[Cañas *et al.*, 2007][Lobato, 2005].

Sin embargo `jde.c` presenta una serie de puntos críticos sensibles a ser mejorados. El objetivo de este proyecto fin de carrera será crear `jdeneo.c`, una nueva implementación de la arquitectura teórica JDE que pretender solventar las mayores deficiencias de `jde.c`.

Para tratar de explicar cómo se ha llevado a cabo este proyecto fin de carrera presentaremos esta memoria en capítulos. En primer lugar describiremos las motivaciones y objetivos, fijando los requisitos que debe cumplir `jdeneo.c`. En segundo lugar, en el capítulo de entorno y plataforma de desarrollo, se analiza en detalle las herramientas *software* que se han empleado a lo largo de la implementación. Posteriormente encontramos un capítulo dedicado a la descripción informática, en el que se cuenta en profundidad la solución adoptada para llevar a cabo los requisitos impuestos. En el siguiente capítulo comprobaremos experimentalmente el comportamiento de la infraestructura desarrollada mostrando ejemplos que utilicen las herramientas que provee `jdeneo.c`. Finalizaremos la memoria describiendo las conclusiones y las líneas futuras de trabajo sobre este proyecto.

Capítulo 2
Objetivos

Una vez presentado el contexto en el que se sitúa este proyecto, fijaremos los objetivos propuestos. En primer lugar describiremos los problemas que han servido de motivación. Posteriormente estableceremos una serie de requisitos, unos impuestos por la arquitectura teórica y otros por las especificaciones de este proyecto. En último lugar se mostrará la metodología y el plan de trabajo seguido.

2.1. Descripción del problema y objetivos

Para el desarrollo de este proyecto hemos partido de `jde.c` 3.4, una implementación en `C` de la arquitectura cognitiva JDE, que se ha implementado sobre una plataforma *hardware* y *software* concretas:

Plataforma hardware

El robot utilizado como referencia para `jde.c` fue el Pioneer 2DXE (Ver figura 2.1). Para acceder a todos los recursos del robot, `jde.c` proporciona una API que permite crear aplicaciones sobre él. Este robot dispone inicialmente de un equipo sensorial básico que consta de sensores de contacto, sensores de ultrasonidos y un par de odómetros asociados a las ruedas motrices. Incluye, así mismo, los motores que permiten al robot moverse por el entorno. Finalmente, contiene un microprocesador Siemens 88C166 que facilita las medidas de los sensores y permite materializar los comandos de actuación para la base motora, todo ello a través de un puerto serie para comunicarse con los ordenadores.

Posteriormente se le añadió otro sensor de distancia a obstáculos: un sensor laser de la marca Sick. Proporciona 10 barridos por segundo con un margen de error muy inferior respecto a las mediciones de ultrasonidos. Al igual que el microcontrolador del robot, presenta sus lecturas a través de un puerto serie.

Además, se le ha dotado de un cuello mecánico (Unidad *pantilt*) junto con dos cámaras. El cuello mecánico, de la casa Directed Perception[1], permite gracias a sus dos motores mover a voluntad las cámaras horizontal y verticalmente, todo ello a través del puerto serie. Las camaras que actualmente contiene el robot son un par estéreo de cámaras *firewire*, modelo Isight[2] de Apple.

[1] http://www.dperception.com/
[2] http://www.apple.com/isight/

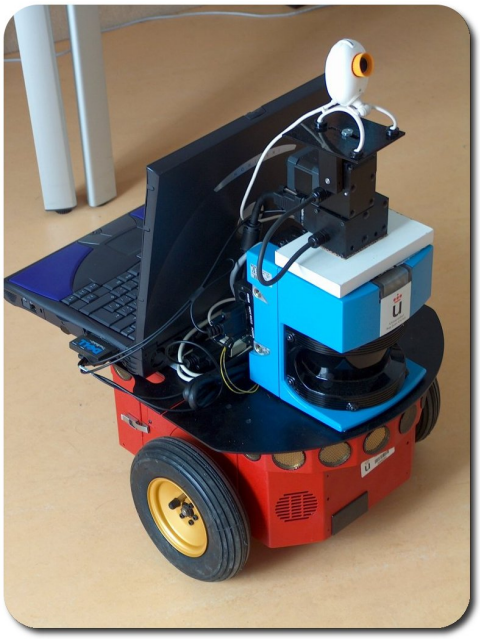


Figura 2.1: Robot Pionner 2DXE

Plataforma software

La implementación `jde.c` está desarrollada en el lenguaje de alto nivel `C`, y funciona sobre el sistema operativo libre `GNU/Linux` (Si bien Linux no es un sistema operativo de tiempo real, ha resultado ser suficientemente ágil para los requisitos de las aplicaciones llevadas a cabo). Emplea la biblioteca `XForms` para generar visualización en forma de ventanas, facilitando la interacción y depuración de la aplicación. El funcionamiento de los esquemas está fuertemente ligado a la biblioteca `pthreads`, con las que se logra que cada esquema funcione como una hebra.

Para conectarse al robot dispone de dos servidores de acceso remoto, que permiten que la infraestructura no se ejecute necesariamente en el portátil a bordo del robot. Tenemos por un lado el servidor `otos`, que reúne los servicios sensoriales de proximidad como sónares y laser, permite enviar comandos de movimiento a la base motora, y entrega los datos de odometría, tanto posición como velocidad estimada. Por otro lado, `óculo` aúna las funciones asociadas al cuello mecánico y a las cámaras, permitiendo de ese modo mover la unidad *pantilt* a voluntad, y pone a disposición de los clientes el flujo de imágenes obtenidas con las cámaras.

2.1.1. Motivaciones

Si bien la versión 3.4 de jde.c cumple exitosamente su objetivo, también hay funcionalidades de la arquitectura cognitiva que no implementa. De igual modo hace uso de bibliotecas poco idóneas para determinadas tareas, como son el *renderizado* 3D o el manejo de interfaces gráficas. El objetivo de este proyecto será principalmente solucionar todos esos problemas creando una nueva plataforma: jdeneo.c. A continuación se detallan en profundidad las motivaciones de este proyecto

Enlazado estático

Como ya hemos visto, la unidad mínima que dota de lógica a JDE es el esquema. Para esto existe un problema, que es el método en que se enlazan el núcleo de la aplicación y los esquemas. Una de las vías es la empleada por jde.c: enlazado estático en tiempo de compilación. Es decir, cada implementación de jde.c es genuina, puesto que incluye en el núcleo todos los esquemas necesarios para su uso. Este comportamiento implica una serie de consecuencias no deseadas, la más importante es que *cada vez que se deseen modificar algún esquema, se debe recompilar todo el sistema.*

Interfaz gráfica

jde.c hace uso de las bibliotecas gráfica `XForm`, basada en `XLib` para el proyecto `X Window System`. Contiene un amplio conjunto de objetos gráficos, tales como botones, barras de desplazamiento, menús, etc. Además, es muy fácilmente extensible. Su funcionamiento se basa, al igual que la mayoría de *toolkits* gráficos, en conexión de eventos a manejadores.

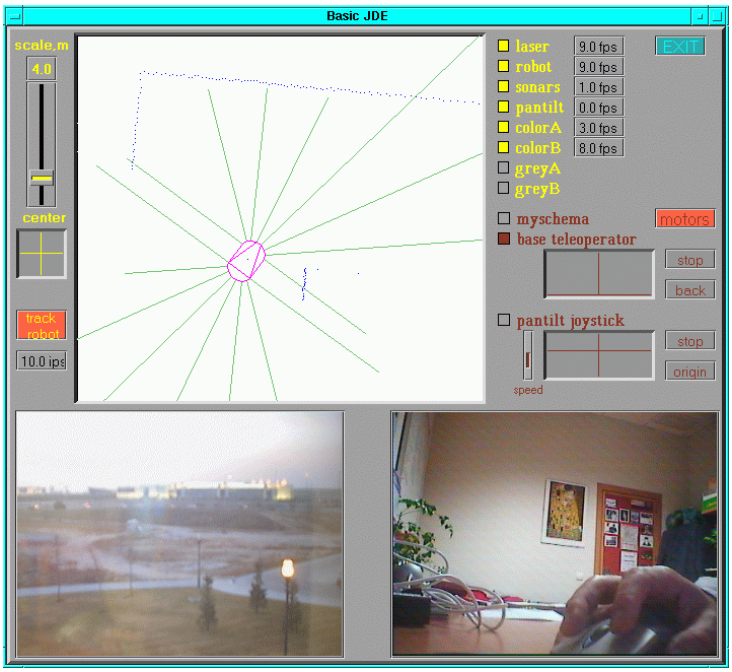


Figura 2.2: jde.c 3.4 haciendo uso de las bibliotecas XForms

Hace un tiempo XForms era muy usado, especialmente en aplicaciones gráficas de Sun Microsystems. Fue un método intuitivo y sencillo de dotar de interfaz a aplicaciones en C. Sin embargo poco a poco ha caído en desuso, debido a diversos factores:

- No es posible utilizarlo en C++ – A tal efecto se creó la biblioteca `FLTK`.
- El aspecto visual es muy pobre.
- No genera código portable – Depende fuertemente de las `X Window System`.
- Es una biblioteca pesada – Se carga toda la biblioteca cada vez que se usa.

Además de esos motivos, hoy día es un proyecto con muy poca actividad que no goza de una comunidad tan activa como otros *toolkits* gráficos. Debido a todo eso, hemos considerado que las XForms es uno de los puntos flacos de jde.c.

Escenas 3D

jde.c utiliza para el tratamiento de escenas 3D la biblioteca `progeo`. Es una biblioteca desarrollada en C para manejar geometría proyectiva (PROjective GEOmetry). Su principal cometido consiste en:

- Proyectar puntos de un espacio en tres dimensiones a un píxel en una imagen en dos dimensiones.
- Retroproyectar un píxel de una imagen dos dimensiones a una línea en un espacio en tres dimensiones.

Emplea visualización basada en foco de atención (*Focus of attention* – FOA), en la que contamos con la posición del observador, el punto de atención, y el giro.

Ésta biblioteca se ha adaptado muy bien para la programación de aplicaciones de visión computacional. Sin embargo su uso para generar imágenes sintéticas y visualización 3D realiza un consumo intensivo del procesador central, por lo que no permite *renderizar* y analizar escenas ricas en detalles. Además está limitado a la generación de puntos y rectas.

Afortunadamente existen bibliotecas y especificaciones que hacen uso del procesador gráfico (GPU) para realizar muchos de estos cálculos, minimizando el uso de CPU. Además es muy común encontrar en este tipo de bibliotecas un modelo basado en foco de atención.

2.1.2. Objetivos

Como hemos visto, `jde.c` presenta una serie de problemas que hacen que la infraestructura no sea todo lo potente que podría. Este proyecto se propone como principal objetivo construir una nueva plataforma que, además, solucione todos los problemas enumerados anteriormente. Podemos dividir esta nueva plataforma en los siguientes objetivos:

1. Se desarrollará empleando bibliotecas gráficas de última generación. Se rediseñará por completo la interfaz gráfica, así como los puntos de unión entre el núcleo de jdeneo.c y el código encargado de la interfaz. También se apostará por un nuevo diseño de la disposición de los elementos gráficos, buscando la usabilidad de cara al usuario. Como veremos más adelante en 3.1, usaremos la biblioteca GTK.
2. El manejo de esquemas será también rediseñado. Entre las modificaciones lograremos la carga dinámica de esquemas – esto es, carga de esquemas en tiempo de ejecución – o el uso de bibliotecas de bajo nivel para sincronización, bloqueos y comunicación entre ellos. Además no será necesario recompilar la infraestructura al modificar un esquema. Para realizar este objetivo utilizaremos las bibliotecas de propósito general GLib (Ver 3.2).

3. Debe haber visualización 3D de la escena de forma nativa. Si bien la visualización en dos dimensiones existente hasta ahora ha sido suficiente, jdeneo.c pretende ser una aplicación escalable y usable, de modo que por un lado podamos prepararlo para esquemas complejos en los que se requiera interactuar con un mundo en tres dimensiones, y por otro haga más atractiva la visualización. Escogeremos una herramienta que alivie el consumo de CPU, delegando todo el procesamiento posible a la GPU. En concreto elegiremos, como veremos en 3.3 la biblioteca OpenGL.

2.2. Análisis de requisitos

Al tratarse de una implementación de la arquitectura teórica de JDE, hay una serie de requisitos que tenemos impuestos, que a grandes rasgos podemos resumir en:

1. La unidad básica de comportamiento es el esquema.
2. El flujo de ejecución de un esquema es independiente.
3. El flujo de ejecución de un esquema es modulable, iterativo y puede ser activado o desactivado a voluntad.
4. Hay esquemas perceptivos y de actuación.
5. Un esquema de actuación puede estar DORMIDO, ALERTA, PREPARADO o ACTIVO.
6. Un esquema perceptivo puede estar DORMIDO o ACTIVO.
7. Un esquema puede utilizar la funcionalidad de otros para conseguir su objetivo.
8. Los esquemas se estructuran en jerarquías, distinguiendo entre padres e hijos.
9. Las jerarquías se construyen y modifican dinámicamente.

Además, jdeneo.c fija una serie requisitos que deben cumplirse para el correcto funcionamiento de la plataforma.

2.3. Metodología y plan de trabajo

Para el desarrollo de este proyecto se realizarán varios bloques de tareas más o menos independientes entre sí, razón por la cual se perseguirá un modelo de desarrollo iterativo que permita ir añadiendo las nuevas funcionalidades a la aplicación final. Además, también será necesario un modelo flexible, ya que gran parte del trabajo se basará en tecnologías software no conocidas a priori. Por tanto se ha escogido un desarrollo usando un modelo en espiral (Figura 2.3).

Si bien desde un punto de vista general el desarrollo se corresponderá con el modelo en espiral, en cada uno de los bloques se utilizará la metodología conocida como *bottom-up*, en la que desde un principio se especifica cada pequeña parte del sistema en gran detalle, para posteriormente unirlas componiendo un bloque completo.

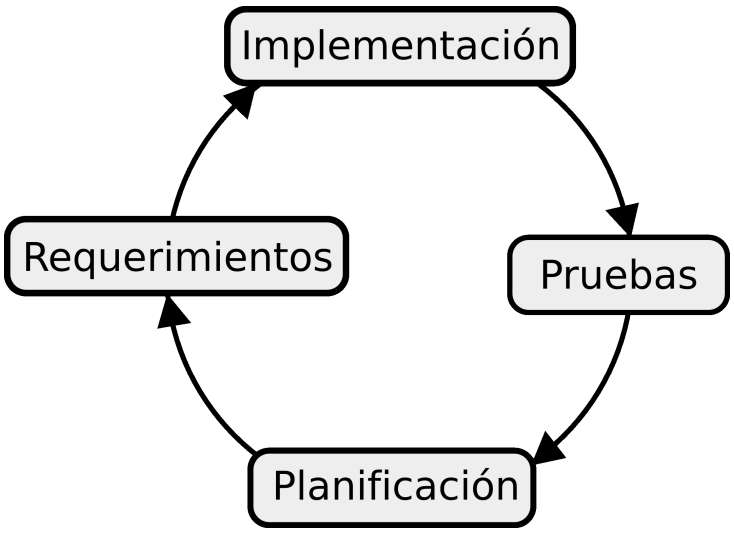


Figura 2.3: Diagrama de iteraciones del modelo de desarrollo en espiral

Por otro lado, la coordinación entre el tutor y el alumno consistirá en un seguimiento detallado de la evolución del alumno, realizando tutorías semanales en las que se fijarán hitos y objetivos a cumplir.

Los hitos asignados componían en ultima instancia bloques correspondientes a iteraciones del modelo empleado. Se ha dividido en las siguientes etapas concretas:

1. Familiarización con la plataforma jde.c. Todo el trabajo se realizará tomando como partida la versión 3.4 de jde.c, por lo que será necesario conocer en detalle el funcionamiento del mismo

2. Interfaz gráfica. En la segunda fase se procederá a modificar por completo las bibliotecas gráficas utilizadas por jde.c. Utilizaremos GTK.

3. Arquitectura de esquemas. Posteriormente se realizará una nueva implementación de la infraestructura que administra los esquemas, empleando nuevos recursos y herramientas. Para esta tarea emplearemos la biblioteca GLib.

4. Visualización 3D. Se dotará de un visor 3D de la escena reconocida por el robot. El motor elegido será OpenGL

5. Casos de uso reales. Finalmente se comprobará el correcto funcionamiento de la infraestructura poniendo en funcionamiento algoritmos clásicos, y probando el correcto funcionamiento de cada parte.

Capítulo 3

Entorno y plataforma de desarrollo

Para poder realizar una nueva plataforma que incluya todas las mejoras mencionadas a la versión 3.4 de `jde.c`, haremos uso de nuevas bibliotecas que sustituyan a sus predecesoras. A la hora de escoger estas bibliotecas se ha perseguido elegir herramientas actuales y estándar, como son GTK+[1] para el diseño de interfaces gráficas, u OpenGL[2] para el *renderizado* de escenas en tres dimensiones.

3.1. GTK+

Existen muchas y variadas alternativas en cuanto a bibliotecas gráficas, por lo que fue necesario realizar una criba inicial. Las dos que se mostraban más potentes y adecuadas eran GTK+ y Qt[3]. Una de las mayores diferencias que existe entre ambas son las facilidades que ofrecen a la hora de programar dependiendo del lenguaje. GTK+ se mostraba más amistoso con C. En cambio Qt parecía más idóneo para C++. Dado que ambas son muy similares en cuanto a funcionalidad y potencia, decidimos emplear GTK+, ya que jde.c está escrito en C principalmente.

Podemos entender GTK+ como una capa de alto nivel que nos permite crear y manejar elementos gráficos para aplicaciones de escritorio (Figura 3.1). Debajo de GTK+ encontramos las GLib – Biblioteca de bajo nivel de propósito general –, y las GDK – Capa intermedia entre el servidor gráfico y GTK+ [Pennington, 1999][Moya *et al.*, 2002].

GTK+ contiene todos los elementos necesarios para diseñar interfaces gráficas. Estos elementos, con los que el usuario puede interactuar, se llaman *widgets*, y son las piezas con las que se construye el aspecto final de la aplicación. Si bien GTK+ está escrito en C, la implementación de *widgets* utiliza conceptos propios de la programación orientada a objetos, como la herencia; cada *widget* hereda todas las funciones y variables de su precedesor. Entre los *widgets* encontramos objetos tales como botones, textos, ventanas, etc.

Sin embargo los *widgets* por sí solos no realizan ninguna tarea. Es el programador el que debe decidir cómo debe reaccionar ante una determinada acción. La forma en que se realiza este mecanismo en GTK+ es muy similar a la de otras implementaciones: Cada *widget* tiene

[1] http://www.gtk.org/
[2] http://www.opengl.org
[3] Qt es una biblioteca multiplataforma GPL para el desarrollo de interfaces gráficas, creada por trolltech. Página oficial: http://www.trolltech.com/

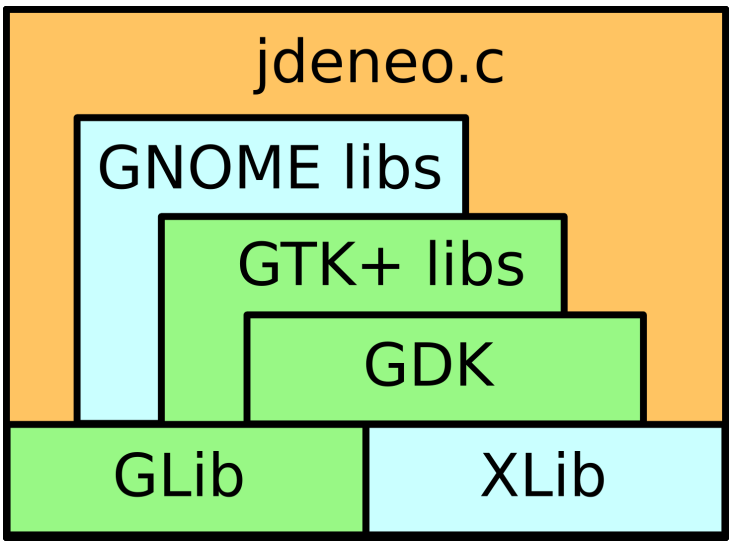


Figura 3.1: Diagrama de interrelación entre jdeneo.c, GTK+ y GLib

una serie de señales asociadas. Estas señales son en general eventos externos que se disparan asíncronamente. El programador puede asociar una función (*callback*) a una determinada señal, de modo que cuando ésta se produzca, se llame de forma automática al *callback*. Por ejemplo, en el caso de un botón una señal sería "clicked", y podríamos asociar una función a ese evento.

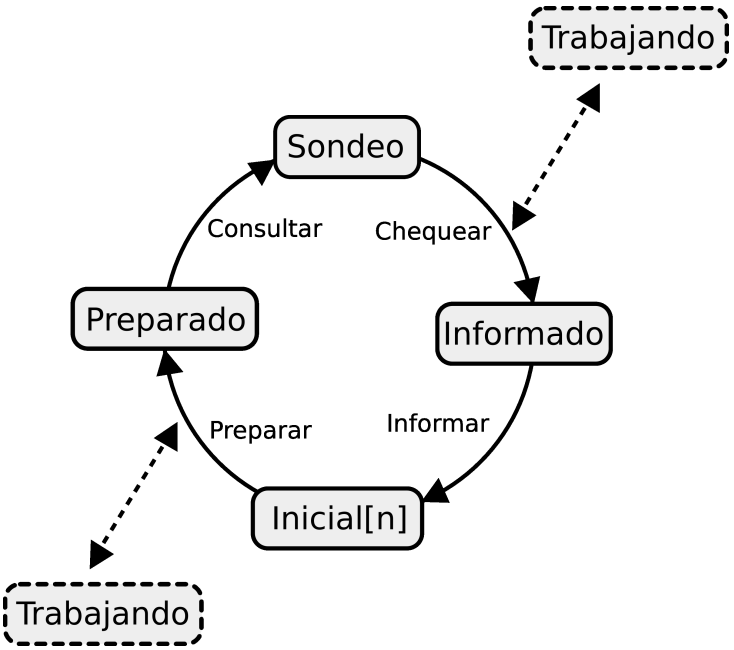


Figura 3.2: Diagrama de estados de un bucle de eventos

GTK+ nos proporciona además un bucle principal (Figura 3.2), que es el encargado de esperar la entrada de datos en la interfaz [GTK, 2007a]. Cuando detecta un evento, se lo notifica a los *widgets* involucrados, y éstos entonces emiten una o más señales y se ejecutan los *callbacks* asociados (si los hubiera). Una vez que han finalizado los *callbacks*, entonces el control vuelve al bucle principal y espera nuevas entradas de datos. Podemos tener múltiples instancias de bucles de eventos, y su funcionamiento es muy similar al de las funciones *idle*; asignamos niveles de prioridad.

El bucle pasa del estado inicial a prepararse para consultar por sondeo el estado de la aplicación. Una vez preparado consulta la información necesaria para el sondeo. Cuando el sondeo ha sido realizado se devuelve el resultado al bucle principal para que quede informado y lance las señales oportunas. Finalmente se retorna a la siguiente iteración del estado inicial. Entre tanto, se puede detener a realizar otras tareas, como por ejemplo atender un *timer* (Ver sección 3.2.3).

Por tanto GTK+ nos permite:

- Crear nuestros propios elementos gráficos
- Definir la disposición de los elementos
- Asociar funciones a eventos
- Crear un bucle de eventos principal

3.1.1. libGlade

El diseño de la disposición de los elementos de la interfaz gráfica es una tarea relativamente sencilla, sin embargo es muy costosa en cuanto a tiempo de codificación, y es más costosa aún si tenemos en cuenta posibles modificaciones del aspecto final de la aplicación. Para hacer más eficiente todo este trabajo existen herramientas que automatizan muchas tareas y nos permiten un diseño visual de layout de la aplicación, es decir, nos aportan un entorno WYSIWYG (What You See Is What You Get).

Para definir interfaces en GTK+ existe Glade[4] (Figura 3.3), y nos permite:

- Crear ventanas
- Añadir *widgets* a las ventanas
- Definir la disposición de todos los elementos
- Modificar las propiedades de cada *widget*
- Asociar funciones a eventos de cada *widget*

Para enlazar la interfaz gráfica creada por Glade con una aplicación en C existen dos vías. Una de ellas es exportando el diseño creado a un fichero con código C, para posteriormente enlazarlo en tiempo de compilación. Esta vía permite tiempos de carga ligeros, pero sin embargo nos permite un menor control sobre el código, ya que no debemos modificar el fuente generado por Glade. La segunda vía guarda el diseño en formato XML, y se carga en la aplicación en tiempo de ejecución. El acceso se realiza utilizando libGlade, que nos aporta una serie de primitivas para cargar el fichero XML, inicializarlo, obtener *widgets* a partir de un nombre, o enlazar de forma automática cada señal con el *callback* que le corresponda [Hemstridge, 2002].

[4] http://glade.gnome.org/

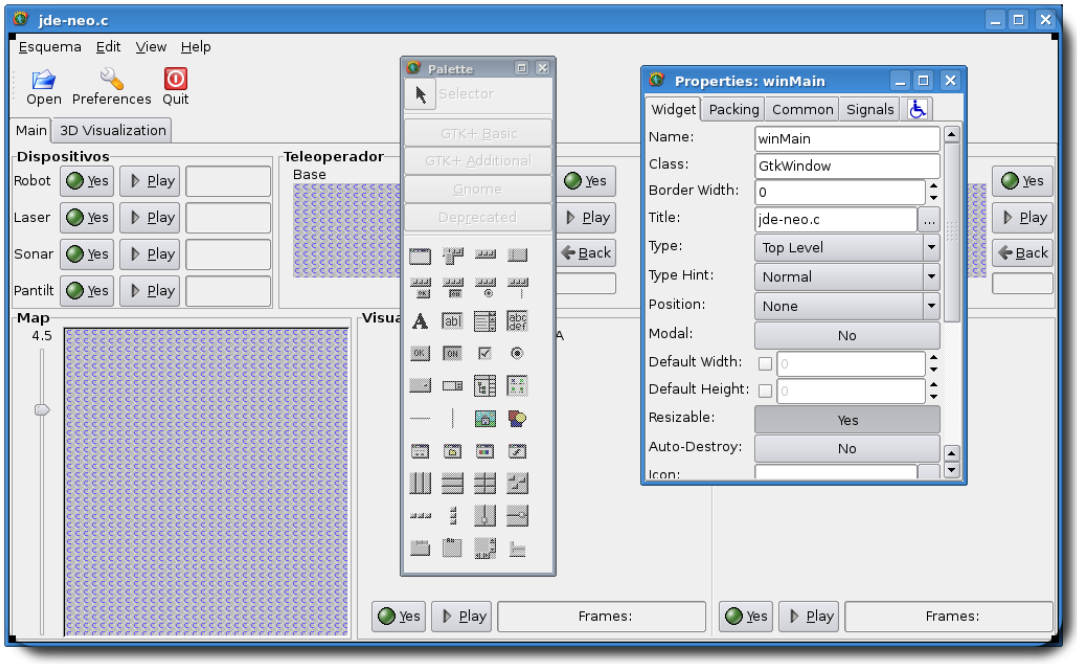


Figura 3.3: Glade y diseño de la interfaz gráfica de una aplicación

3.1.2. libGnomeCanvas

Un *canvas* es un área en el que una aplicación puede *dibujar*. Por lo general los *canvas* permiten añadir polígonos, líneas, puntos y formas básicas. Es una herramienta muy útil ya que permite mostrar de forma gráfica detalles que serían muy complicados de mostrar usando otro modo.

GTK+ no contiene ningún *canvas*, por lo que es necesario recurrir a alguna biblioteca externa para poder insertarlo en una aplicación GTK+. Uno de los más populares es libGnomeCanvas[Can, 2007], parte del proyecto Gnome[5]. libGnomeCanvas proporciona métodos para tratar cada elemento por separado, de modo que, por ejemplo, resulta muy sencillo modificar las coordenadas de un polígono para que se sitúe en otro lugar dentro del *canvas*. Además contiene funciones de conversión de sistemas de unidades, así como *antialiased*, zoom, etc.

Cabe mencionar que libGnomeCanvas, al ser una herramienta muy estándar, se encuentra disponible en la mayoría de distribuciones GNU/Linux.

3.1.3. libGtkGLExt

Del mismo modo que un *canvas* nos permite representar en dos dimensiones una escena, también es muy útil poder hacer una representación de cierta información en tres dimensiones. Una de las especificaciones más estándar para generar aplicaciones que produzcan gráficos en 3D es OpenGL. Sin embargo GDK no proporciona ningún mecanismo para usar y visualizar código OpenGL en una aplicación, por lo que de nuevo es necesario recurrir a bibliotecas externas que permitan incluir visualización 3D usando OpenGL en una aplicación GTK+.

[5]http://www.gnome.org/

Una de esas bibliotecas es libGtkGLExt[6]. Es una extensión de GTK+ que proporciona objetos GDK adicionales que soportan *renderizado* OpenGL en GTK+ [gtk, 2007b]. Al igual que libGnomeCanvas, se trata de una biblioteca muy común por lo que es muy fácilmente instalable.

3.2. GLib

Como hemos visto, GTK+ funciona sobre GLib, una plataforma de propósito general parte del proyecto GTK+. GLib permite, así mismo, realizar aplicaciones multiplataforma[7]. Entre sus principales funcionalidades encontramos:

- Tipos básicos de datos (Ver sección 3.2.3)
- Conversión entre tipos
- Ubicación en memoria
- *Timers* (Ver sección 3.2.3)
- Utilidades de cadenas de caracteres
- Carga dinámica de módulos – GModule (ver sección 3.2.2)
- Hebras – GThreads (Ver sección 3.2.1)
- Un sistema de objetos – GObject
- Análisis de ficheros tipo ini (Ver sección 3.2.3)
- Estructuras de datos complejas – Listas (Ver sección 3.2.3), árboles, pilas, tablas hash, etc.

A continuación describimos brevemente las herramientas de GLib empleadas para el desarrollo de `jdeneo.c`

3.2.1. Paralelismo - Gthread

GLib proporciona una capa de abstracción de hebras. Con las hebras conseguimos paralelismo, de modo que todas ellas se ejecutan concurrentemente. A diferencia de los procesos, las hebras comparten entre sí un mismo espacio de memoria, por lo que es necesario contar con mecanismos de interbloqueo y sincronización entre ellas para que no haya datos corruptos. Todo ésto y más nos lo proporciona GLib.

GLib nos facilita las siguientes herramientas para la programación concurrente:

1. Cerrojos – Mediante `GMutex` podemos realizar bloqueos al área de memoria que deseemos.

[6] http://www.k-3d.org/gtkglext

[7] GLib actualmente funciona en la mayoría de los sistemas operativos basados en UNIX, y en Microsoft Windows.

2. Variables de condición – Usando GCond podemos sincronizar hebras mediante variables de condición.

3. Datos privados – `GPrivate` permite que cada hebra tenga un puntero propio a una variable privada.

4. Hebras – Finalmente tenemos `GThread`, que primitivas para crear y administrar hebras de forma portable

Para el desarrollo de `jdeneo.c` la más importante de todas ellas ha sido `GThread`. Su funcionamiente es muy similar al de la biblioteca pthreads. Permite lanzar hebras a voluntad de forma sencilla estableciendo parámetros, prioridad de ejecución, etc. Así mismo la detención de una hebra puede ser o bien voluntaria – la hebra decide finalizar su ejecución – o bien forzada – otra hebra principal decide detenerla.

3.2.2. Carga dinámica - GModule

A menudo se desea poder añadir nuevas funcionalidades a una aplicación sin necesidad de que esa funcionalidad vaya incluida de forma nativa. De este modo, se puede publicar por un lado la aplicación principal, y por otro extensiones – *plugins*[8]. Cuando hablamos de carga dinámica nos referimos a la posibilidad de enlazar una aplicación con una extensión en tiempo de ejecución.

A tal efecto GLib contiene GModule, que básicamente es una serie de primitivas que permiten la carga dinámica de forma portable. No es de extrañar que su interfaz sea muy similar a la utilizada en la biblioteca dlopen, ya que en sistemas UNIX en última instancia está utilizando precisamente dlopen. A grandes rasgos podríamos enumerar las siguientes operaciones que nos facilita GModule:

1. Apertura de un módulo – Abre un módulo a partir de la ruta en la que se encuentra ubicado en el árbol de directorios. Es en este momento cuando especificamos cómo deseamos cargarlo; los símbolos pueden o no resolverse sólo cuando son llamados, y pueden o no ser añadidos al espacio global de nombres.

2. Obtener un símbolo – A partir de un módulo y el nombre de un símbolo podemos obtener un puntero a dicho símbolo.

3. Cierre de un módulo – Finalmente, podemos cerrar el módulo liberando los recursos ocupados.

Así mismo, los módulos pueden implementar unas determinadas funciones tanto para la apertura como para el cierre, de modo que si existen serán llamadas de forma automática.

3.2.3. Otras herramientas de GLib

Para el desarrollo de jdeneo.c se han utilizado muchas otras herramientas de las que proporciona GLib. A continuación se describen brevemente.

[8]Un ejemplo muy conocido de *plugins* son las extensiones disponibles para el navegador Firefox.

Timers

Existen situaciones en las que quizás deseemos que una función se ejecute de manera automática cada cierto tiempo. GLib nos proporciona *timeouts*, que son los encargados de llamar cuando proceda a la función deseada. El *timeout* puede finalizar de dos modos: o bien la función llamada por un *timeout* decide cuándo desea finalizar su ejecución, o bien se puede provocar la parada desde el programa principal. A diferencia de las hebras, una función ejecutada desde un *timeout* no se ejecuta de forma concurrente, sino que el bucle principal detiene su ejecución hasta que la función ha finalizado.

Ficheros .ini

Las aplicaciones a menudo contienen un diálogo desde el cual se puede modificar la configuración. En general esa configuración suele ser única para cada usuario del sistema, y además no es volátil. La mayoría de las soluciones pasan por almacenar en disco uno o más ficheros con la configuración de cada usuario. Para ello existen convenios, como es el uso de ficheros tipo ini, o emplear estructuras más complejas utilizando ficheros XML.

GLib contiene una biblioteca para el análisis de ficheros de valor-de-clave (Ficheros .ini). Esto es, que a cada clave se le asigna un valor – p.ej. impresora=HP1210. Utilizando este analizador accederemos a cualquier campo de forma sencilla, y podremos modificarlo sin preocuparnos por cómo están organizados los datos, el acceso al fichero, o manejar cadenas de caracteres.

Listas enlazadas

GLib contiene también estructuras de datos avanzadas, como son listas, árboles, tablas hash, o conjuntos. Para el desarrollo de `jdeneo.c` hemos empleado listas enlazadas, ya que en determinadas situaciones es deseable poder almacenar un número indeterminado de elementos de un cierto tipo.

GSList es la biblioteca que nos aporta GLib para el manejo de listas. Éstas contienen tan solo un puntero al siguiente registro, y un puntero genérico para que enlace a los datos almacenados que se desean almacenar en el registro. Desde GSList podemos crear de forma sencilla una nueva lista, unir dos listas, insertar, modificar y borrar elementos, copiar una lista, calcular longitud, liberar espacio en memoria, etc.

Tipos de datos básicos

GLib tiene creada una capa para abstraer de los datos propios de cada arquitectura y plataforma. De ese modo las aplicaciones son fácilmente portables. Éstos datos son por ejemplo gint8, guint16, etc. Además, se definen tipos que no existen en C estándar, como son gboolean o gsize. También contiene tipos básicos de números enteros y reales, como por ejemplo gint, guint, gchar, gshort, etc.

3.3. OpenGL

Para el desarrollo de aplicaciones que utilicen visualización en 3D se emplean lenguajes de bajo nivel que aseguren una API multiplataforma y unas primitivas básicas para lograr cualquier objetivo. OpenGL (Open Graphics Library) es probablemente la definición más estandarizada en cuanto a lenguajes de modelado en 3D – Adicionalmente también resulta muy potente para procesar datos en 2D. Creado y mantenido por Silicon Graphics[9] a principios de los 90, actualmente se utiliza ampliamente en la industria de los videojuegos, en realidad virtual y en aplicaciones de diseño asistido (CAD) entre otros.

Básicamente, OpenGL es una especificación, entendiéndolo como un convenio que describe un conjunto de funciones y el comportamiento que deben cumplir las plataformas que lo cumplan [Neider *et al.*, 1993]. A partir de esta especificación, los fabricantes de hardware crean implementaciones – Bibliotecas creadas para coincidir con las especificaciones de OpenGL, haciendo uso de aceleración hardware siempre que sea posible. Existen implementaciones que hacen uso de aceleración hardware para la mayoría de los sistemas operativos y para muchas videoconsolas como Wii o PlayStation 3. Por otra parte existe una implementación que sólo hace uso de software; la biblioteca de código abierto Mesa[10].

En el diseño de OpenGL se persiguen dos objetivos principales:

1. Ocultar la complejidad de crear interfaces para las diferentes aceleradoras 3D, presentando al programador una única API uniforme

2. Ocultar las diferentes capacidades de las plataformas hardware, exigiendo que todas las implementaciones soporten todo el conjunto de funcionalidades de OpenGL (haciendo uso de emulación software si fuera necesario)

Entre las operaciones básicas que contiene OpenGL encontramos primitivas para crear puntos, líneas o polígonos. Todo esto se hace por un pipeline gráfico (Ver figura 3.4)conocido como la máquina de estados de OpenGL. Muchos de los comandos OpenGL configuran el modo en el que el pipeline procesa las primitivas.

Alrededor de OpenGL existe una serie de bibliotecas auxiliares que extienden su funcionalidad.

Entre esas bibliotecas tenemos, por ejemplo, GLU (OpenGL Utility Library), que es una biblioteca que ofrece funciones de más alto nivel que las proporcionadas por OpenGL. En general se distribuye junto con el paquete base de OpenGL. Entre sus funcionalidades encontramos correspondencias entre coordenadas de la pantalla y coordenadas del mundo, generación de texturas mipmaps[11], superficies cuadráticas, interpretación de errores de código de OpenGL, etc.

[9] http://www.sgi.com/

[10] http://www.mesa3d.org/

[11] Un mipmap es una colección de imágenes optimizadas para acompañara una textura principal, para procurar minimizar el tiempo de *renderizado*.

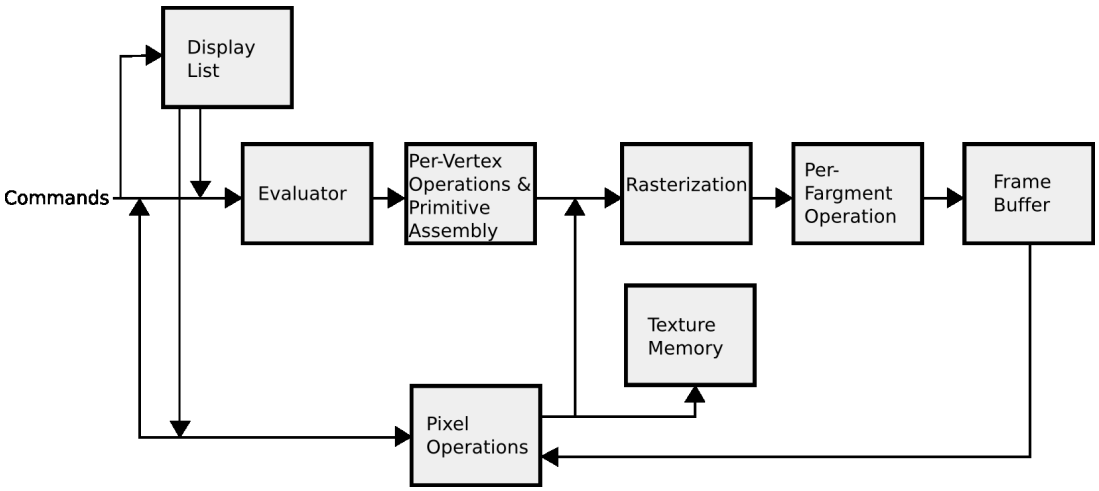


Figura 3.4: Pipeline gráfico de OpenGL

Por otro lado nos proporciona también GLUT (OpenGL Utility Toolkit), una biblioteca que principalmente interpreta llamadas de entrada/salida con el sistema de ventanas usado. Algunas de las funciones incluidas son la definición y control de ventanas, monitorización de eventos de teclado y ratón, etc. También se proveen funciones para *renderizar* una serie de objetos geométricos básicos como cubos o esferas. Su objetivo es facilitar el desarrollo de aplicaciones multiplataforma, y sobre todo facilitar el aprendizaje de OpenGL sin tener conocimientos de complejas APIs dependientes del sistema para el manejo de ventanas.

3.3.1. Blender

El modelado de figuras reales puede resultar bastante complejo programando OpenGL a mano, ya que sus primitivas son de muy bajo nivel y haría el trabajo de modelarlo muy costoso. Se ha buscado, por tanto, otro método que asistiera en el diseño 3D del robot para posteriormente exportarlo a código C en OpenGL. La solución ha sido utilizar Blender[12], una aplicación *open source* para modelado y *renderizado* de escenas en 3D (Ver figura 3.5). OpenGL utiliza la aplicación de trazado de rayos YafRay (**Y**et **A**nother **F**ree **Ray**tracer), una aplicación de trazado de rayos que emplea un lenguaje de descripción de escenas en XML.

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

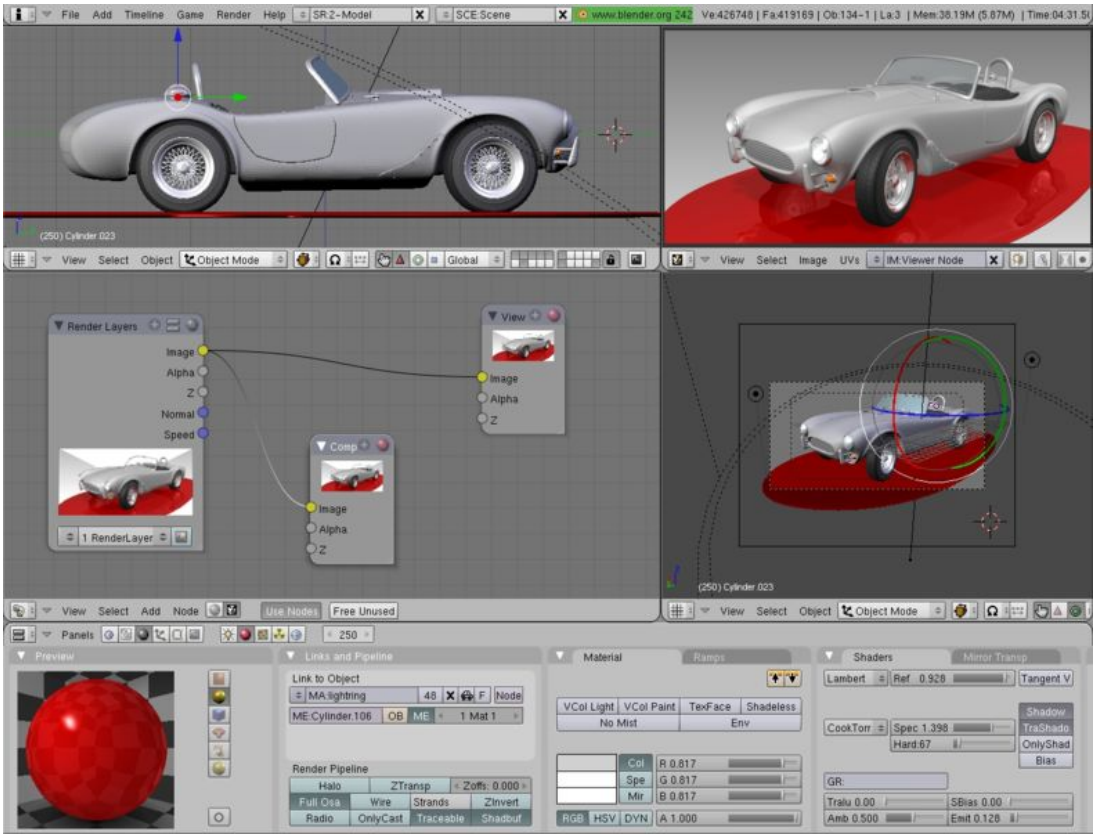


Figura 3.5: Blender: Una aplicación de *renderizado* y modelado en 3D

Capítulo 4

Descripción informática

A lo largo de este capítulo describiremos la solución desarrollada para alcanzar el objetivo planteado en el capítulo 2, empleando las herramientas que se presentaron en el capítulo 3. En primer lugar mostraremos de forma general el diseño realizado, de modo que se comprenda cómo encajan las distintas partes para formar una arquitectura coherente. Después se describirá en detalle el funcionamiento de todos los mecanismos relacionados con los esquemas. En tercer lugar presentaremos la solución adoptada para generar la escena en 3 dimensiones empleando OpenGL. Finalmente describiremos en profundidad el diseño y programación de la interfaz gráfica.

4.1. Diseño General

`jdeneo.c` es un nuevo proyecto nacido a partir de `jde.c` 3.4, motivo por el cual hereda gran parte de su infraestructura, organización y *drivers* de acceso a los dispositivos físicos. Sin embargo, se ha desarrollado una nueva plataforma desde cero, tratando de emplear, como ya hemos visto, herramientas de última generación en la medida de lo posible. Además, durante todo el diseño se ha mantenido intacta la API de variables que `jde` ofrece a los esquemas para acceder a los recursos del robot.

El acceso a los dispositivos físicos del robot se realiza por distintas vías dependiendo del tipo de dispositivo:

- Imágenes y cuello mecánico. Para obtener esta información podemos emplear o bien el servidor `oculo`[1] o bien acceso directo a cada dispositivo. En el caso de las imágenes, tenemos varias fuente de estímulos: video4linux, firewire, imágenes estáticas o el puerto USB.
- Al resto de sensores y actuadores se accede por una sola vía: a través del servidor otos[2].

A lo largo del desarrollo de jdeneo.c, se ha empleado el simulador SRIsim para poder comprobar un correcto comportamiento de la plataforma sin necesidad de tener conexión con el robot real.

[1] Oculo es un servidor que provee acceso al cuello mecánico y a las imágenes a través de una interfaz de mensajes.

[2] El servidor otos provee acceso de lectura de láser, sónar, *bumpers* y *encoders*. Ofrece también una interfaz de mensajes para controlar la base motora del robot

4.1.1. Interfaz de Programación de Aplicaciones – API

Una de las principales características de `jde.c` es el método en el que presenta los recursos del robot al programador de esquemas: variables compartidas. Son por tanto variables accesibles en todo momento a cualquier esquema, ya sea para escritura (si se trata de actuadores) o para lectura.

`jdeneo.c` ha heredado este comportamiento de su predecesor. Además se mantiene una misma API de variables accesible a cualquier esquema (Ver tabla 4.1).

Nombre	Recurso
`laser`	Array que contiene 180 valores correspondientes a la distancia medida por el sensor laser en 180 grados.
`us`	Array de 180 valores correspondientes a la distancia medida por los 16 sónares repartidos por la base del robot.
`robot`	Array que contiene la posición del robot de acuerdo al sistema de referencia inicial.
`v`	Velocidad frontal de los motores.
`w`	Velocidad angular de los motores.
`colorA`	Array que contiene una instantánea de la cámara A.
`colorB`	Array que contiene una instantánea de la cámara B.
`pan_angle`	Ángulo deseado para el pan del cuello mecánico.
`tilt_angle`	Ángulo deseado para el tilt del cuello mecánico.
`longitude`	Ángulo medido en pan del cuello mecánico.
`latitude`	Ángulo medido en tilt del cuello mecánico.
`longitude_speed`	Velocidad deseada del pan del cuello mecánico.
`latitude_speed`	Velocidad deseada del tilt del cuello mecánico.

Cuadro 4.1: API de `jdeneo.c`

4.1.2. Organización del código

A lo largo del desarrollo se ha tratado de separar el código en distintos ficheros fuente, de modo que funcionalidades distintas se encuentren en ficheros diferentes. Así, por ejemplo, tenemos en un fichero el núcleo, en otro la interfaz gráfica y en otro el manejo de esquemas, todos ellos trabajando en conjunto para construir el sistema final (Ver figura 4.1).

En general, los principales bloques de código de jdeneo.c son:

- `jde.c` – Núcleo de jdeneo.c. Contiene la API por la cual los esquemas acceden a los recursos del robot. Contiene también los drivers necesarios para conectarse al simulador y/o al servidor remoto, o para manejar los dispositivos físicos (por ejemplo, sónar, laser o motores).Es también el encargado de inicializar y conectar el resto de bloques entre sí, y de cargar la configuración.

- `schemas.c` – Es el encargado de almacenar y proveer la infraestructura para crear y eliminar esquemas en tiempo de ejecución. Permite así mismo acceder a gran parte de los

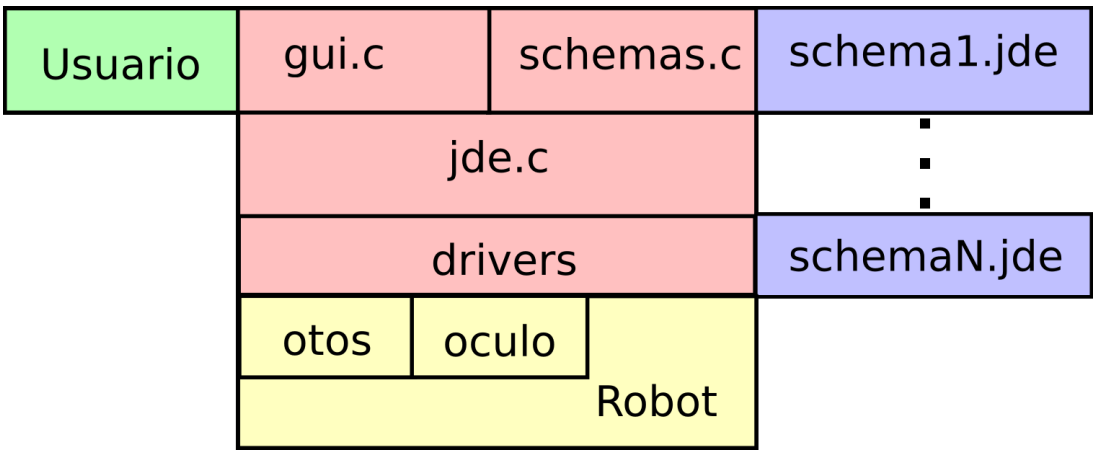


Figura 4.1: jdeneo.c: Diagrama de relación de las principales partes de jdeneo.c

recursos de un esquema.

- `gui.c` – Prácticamente todo el manejo de la interfaz gráfica se gestiona y controla desde `gui.c`. Contiene por tanto todo el código de carga de la interfaz, los manejadores de señales, etc. Además, incluye el código de acceso y modificación a las preferencias del usuario, y la visualización 3D de la escena reconocida por el robot.

4.2. Esquemas

Uno de los principales bloques de este proyecto ha sido la infraestructura de esquemas. Como ya vimos en 2.2, entre los objetivos propuestos tenemos hacer uso de nuevas herramientas cumpliendo los requisitos fijados por la arquitectura cognitiva JDE. Debido a esto, se han rediseñado por completo todos los mecanismos subyacentes a los esquemas.

Por un lado, tendremos carga dinámica de esquemas empleando `GModule`. Por otro lado renovaremos la interfaz de programación de los esquemas, fijando una serie de primitivas básicas. También se creará un nuevo mecanismo de comunicación entre esquemas, de nuevo empleando `GModule`. Finalmente rediseñaremos el funcionamiento de la jerarquía entre esquemas.

4.2.1. Carga dinámica de esquemas

Para poder realizar carga dinámica de esquemas haremos uso de `GModule`. Para ello tendremos que realizar decisiones sobre el tipo de enlazado que realizaremos, y estudiar el funcionamiento de las primitivas de `GModule`. En esta sección entenderemos módulo y esquema como sinónimos.

Política de enlazado

GModule contiene una serie de flags para determinar el tipo de enlazado que se realizará.

Flag	Activado	Desactivado (Por defecto)
G_MODULE_BIND_LAZY	Los símbolos se resuelven sólo cuando van a ser usados	Todos los símbolos se atan cuando el módulo se carga
G_MODULE_BIND_LOCAL	Los símbolos no se tienen que añadir al espacio global de nombres	Los símbolos se añaden al espacio global de nombres

En este contexto deseamos que no puedan solaparse los nombres de los símbolos, de modo que esquemas distintos puedan tener, por ejemplo, una variable con el mismo nombre sin que ello genere un conflicto. Por tanto los símbolos no deben añadirse al espacio global de nombres – G_MODULE_BIND_LOCAL activado. Una vez fijado este requisito, la decisión del momento en que se resuelvan los nombres no afecta de forma relevante el funcionamiento del sistema. Hemos decidido emplear la opción por defecto – G_MODULE_BIND_LAZY desactivado.

Funcionamiento con GModule

Como ya vimos en 3.2.2, el funcionamiento de GModule es muy similar al de otras bibliotecas de enlazado dinámico, como por ejemplo dlopen. En primer lugar jdeneo.c carga el módulo. Posteriormente obtienen los símbolos deseados del esquema. Finalmente, se cierra el esquema liberando todos sus recursos.

Veamos con un ejemplo los mecanismos empleados en jdeneo.c para realizar la apertura de un esquema.

```
GModule *esquema;
gchar *ruta = "/usr/share/jdeneoc/schemas/basic.jde";
...
esquema = g_module_open (ruta, G_MODULE_BIND_LOCAL);
```

Vemos, por tanto, que es en este momento cuando especificamos los *flags* sobre el modo de enlazado que se realizará. Por otra parte podemos comprobar cómo a partir de la ruta absoluta, g_module_open() accede al recurso en disco y lo carga en memoria.

La resolución de símbolos (Variables y funciones de los esquemas) será igualmente sencilla. Necesitaremos proporcionar tres parámetros a la función encargada de ello, g_module_symbol(); (1) el puntero al módulo, (2) el nombre del símbolo y (3) un puntero a la variable en la que deseamos guardar el puntero al símbolo importado. Adicionalmente, g_module_symbol(), nos devuelve un booleano informando de si ha habido error o no. Veámoslo con un ejemplo:

```
gchar **schema_path;
gchar *id_schema = "basic";
GModule *schema;

... /* Se realiza la apertura del esquema */

if (!g_module_symbol(schema, id_schema, (gpointer *) &schema_path))
```

```
    g_error ("error: %s", g_module_error ());
```

En este ejemplo, estamos accediendo a la variable compartida *schema_path* del esquema cuyo identificador es "basic". En caso de que el símbolo no se haya resuelto correctamente mostramos un error y finalizamos la ejecución de `jdeneo.c`[3].

Finalmente, para cerrar un esquema previamente abierto, tan solo tendremos que realizar una llamada a `g_module_close()`, que nos devolverá un `booleano` informando de si se ha cerrado correctamente o ha habido algún error:

```
GModule *esquema;

... /* Se realiza la apertura del esquema */

if (!g_module_close(esquema))
   g_warning("warning: %s", g_module_error ());
```

En este caso, si ha habido error cerrando el esquema, informamos por consola al usuario del mismo mediante `g_warning()`[4].

4.2.2. Interfaz de programación de esquemas

Como hemos visto, la aplicación robótica se construye como una colección de esquemas. Éstos a, su vez, se construyen como bibliotecas dinámicas que posteriormente se utilizan como *plugins*. Surge por tanto un problema; ¿Cómo resuelven jdeneo.c en tiempo de ejecución los nombres usados para las funciones y las variables necesarias?. Para solucionarlo hemos creado una API (Application Programming Interface – Interfaz de Programación de Aplicaciones) que fija de antemano un convenio de nombres a emplear. A continuación se detalla cada una de las funciones y variables que todo esquema debe tener para integrarlo en la infraestructura

Identificador	Tipo	Función
`schema_path`	Variable	Ruta absoluta del esquema.
`schema_id`	Variable	Cadena que identifica unívocamente al esquema.
`schema_interval`	Variable	Valor en milisegundos que determina la duración de cada iteración.
`schema_startup`	Función	Inicialización y activación del esquema.
`schema_suspend`	Función	Detención del esquema.
`schema_gui_startup`	Función	Creación e inicialización de la interfaz gráfica del esquema.
`schema_gui_suspend`	Función	Ocultación y liberación de la interfaz gráfica.

Cuadro 4.2: API de esquemas

[3] En GLib, la función `g_error()` muestra por pantalla un mensaje de error, y posteriormente causa la finalización de la aplicación.

[4] A diferencia de `g_error()`, la función `g_warning()` no finaliza la ejecución del programa.

schema_path

Los esquemas a menudo irán acompañados de uno o más ficheros, como por ejemplo el fichero xml que define su propia interfaz gráfica en glade, o un mapa de bits. El diseño acordado propone que cada esquema localice sus propios recursos a partir de una ruta relativa, de este modo cambiar de posición en el árbol de directorios al esquema resulta algo trivial. Sin embargo los esquemas son construidos como bibliotecas dinámicas, motivo por el cual a priori no pueden conocer la ruta absoluta en la que están alojados, y por tanto no pueden acceder a sus recursos.

La solución a este problema pasa por jdeneo.c, el cual tiene una lista de directorios que consulta al ser arrancado para localizar esquemas, por tanto jdeneo.c sí conoce la ruta completa en la que está alojado cada uno de ellos. El modo de solucionar este problema ha consistido en que jdeneo.c informe a cada esquema de su ruta a través de una variable compartida; schema_path.

```
G_MODULE_EXPORT gchar* schema_path;
```

schema_id

Los esquemas deben poder ser referenciados de forma unívoca. Por ejemplo, para poder registrarlos en el censo, o para que otro esquema pueda importar un símbolo suyo. La solución escogida es usando identificadores que son cadenas de caracteres. En consecuencia, se hace necesario que todo esquema defina la variable `schema_id`.

Debido a esta funcionalidad, el sistema prohíbe la existencia de dos esquemas con un mismo identificador. Cuando se da el caso de cargar un esquema con un identificador que está siendo usado por otro esquema ya cargado, entonces se informa mediante un warning y se suspende la carga.

```
G_MODULE_EXPORT const gchar* schema_id = "basic";
```

schema_interval

Los esquemas contienen una función encargada de realizar la tarea principal. Esta función se ejecuta cada cierto tiempo, y siempre y cuando se cumplan determinadas precondiciones. De modo que hay casos en los que se necesita una aproximación lo más cercana posible a tiempo real (por ejemplo, comprobar si un evento crítico se ha cumplido), y en otros casos la ejecución de la función puede ser demorada a una vez cada 10 segundos (por ejemplo, comprobar la temperatura).

Para establecer el tiempo transcurrido el cual debe realizarse la siguiente iteración del esquema usaremos `schema_interval`, asignándole un valor en milisegundos.

```
G_MODULE_EXPORT guint schema_interval = 1000;
```

Para poder llamar cada cierto tiempo a la función principal hemos utilizado los *timeouts* que nos proporciona GLib. Estos *timeouts* reciben tres parámetros:

1. tiempo en milisegundos
2. función que debe ser llamada
3. puntero a una variable en la que notificar posibles errores

Esta función nos devuelve un entero con el identificador del *timeout* recién creado, para permitir detener a voluntad la ejecución del esquema.

Por otra parte, la función principal devuelve siempre tras su ejecución un valor booleano. En caso de ser `TRUE`, la ejecución del timeout continúa y volverá a ser llamada. En caso de ser `FALSE`, se provoca la finalización del timeout.

La interfaz de jdeneo.c también permite modificar a otros esquemas y al usuario el valor de `schema_interval` de cada esquema.

schema_startup

Esta función, a diferencia de su homónima en jde.c, realiza varias funciones.

Por un lado es llamada automáticamente cada vez que se carga un esquema. Realiza por tanto el código de inicialización del esquema. Una tarea típica en este contexto es el exportado de variables.

Por otro lado, y esta es la parte diferente a jde.c, se encarga de activar y reactivar la ejecución del esquema, esto es, arrancar el *timeout* que lleva a cabo la tarea principal.

```
G_MODULE_EXPORT void schema_startup (void);
```

schema_suspend

Del mismo modo que podemos activar la ejecución de un esquema, podemos también desactivarlo. En términos generales la función de `schema_suspend` es la de finalizar la ejecución del *timeout* iniciado en `schema_startup`.

```
G_MODULE_EXPORT void schema_suspend (void);
```

schema_gui_startup

Al igual que la infraestructura tiene su propia interfaz gráfica, los esquemas también pueden disponer de su propio interfaz para que el usuario visualice e interactúe con el esquema. Esta

funcionalidad se ha construido como algo opcional, de modo que un esquema no tiene necesariamente que contener su propia interfaz.

Adicionalmente, jdeneo.c debe poder activar o desactivar a voluntad la interfaz del esquema, de modo que en un momento dado es posible tener en funcionamiento un esquema pero sin visualizar necesariamente su interfaz gráfica.

Como ya hemos explicado, el funcionamiento del *toolkit* gráfico GTK+ está basado en eventos. Podemos atar cada uno de ellos a una función manejadora. En este contexto, las funciones manejadoras son propias del esquema (no pueden ni deben ser exportadas al resto de la aplicación), por lo que debe ser el propio esquema el encargado de realizar las operaciones necesarias para asociar cada señal a su manejador. Este código va incluido en `schema_gui_startup`, y tan sólo debe ejecutarse una vez.

```
G_MODULE_EXPORT gboolean schema_gui_startup ();
```

schema_gui_suspend

Como ya hemos visto, la interfaz gráfica de un esquema debe poder ocultarse al usuario, de modo que el esquema esté funcionando en *segundo plano*. A tal efecto está la función `schema_gui_suspend`, que básicamente se encarga de ocultar la ventana principal del esquema. Además, al no poder interactuar con la interfaz, no se dedican recursos computacionales a la visualización de ese esquema.

```
G_MODULE_EXPORT gboolean schema_gui_suspend ();
```

4.2.3. Comunicación entre esquemas

Del mismo modo que un esquema se comunica con el núcleo de jdeneo.c a través de una API preestablecida, los esquemas también se pueden comunicar entre ellos a través de punteros a *variables compartidas*. Para ello jdeneo.c proporciona una serie de primitivas que permite que los esquemas puedan importar y exportar variables. Entre tanto, jdeneo.c almacena un censo con todas las variables exportadas.

Censo de variables

Para almacenar los punteros a las variables compartidas por los esquemas, jdeneo.c tiene un censo en el que se guarda toda la información necesaria para localizar unívocamente una variable deseada. Este censo se ha materializado como una estructura dinámica, de modo que podemos almacenar un número máximo indefinido de elementos. Cada uno de estos elementos contiene un registro en el que se almacena metainformación de la variable, como por ejemplo el nombre de su esquema, y se almacena también el propio puntero a la variable compartida.

```
typedef struct _schema_vars schema_vars;
```

```
struct _schema_vars {
   gchar *id;
   gchar *varname;
   void  *var;
};

extern GSList *vars;
```

Vemos pues que la información almacenada es:

- varname – Identificador propio de la variable.
- id – Identificador del esquema.
- var – Puntero a cualquier tipo de dato.

Vemos también que la estructura elegida es una lista simple de las que provee GLib (GSList). Las tareas como añadir, localizar, modificar y eliminar elementos del censo de variables se harán haciendo uso de las primitivas propias de GLib para manejar listas de tipo GSList.

Importar/Exportar variables

jdeneo.c facilita la tarea de importar y exportar variables externas. Para ello se han creado dos funciones accesibles para cualquier esquema; `getSchemaVar()` y `putSchemaVar()`.

`putSchemaVar()` recibe como parámetro el identificador del esquema, el identificador de la variable, y el puntero a la variable. Esta función crea un nuevo registro al que asigna los valores pasados como parámetro, y posteriormente lo inserta en la lista.

```
void putSchemaVar(gpointer var, gchar *varname, gchar *schema);
```

`getSchemaVar()` recibe dos parámetros; el identificador de la variable y el identificador del esquema. Después recorre la lista hasta localizar la variable deseada. Finalmente devuelve el puntero a esa variable.

```
void *getSchemaVar(gchar *id, gchar *varname);
```

En la figura 4.2 vemos cómo se interrelacionan todos los elementos implicados en la compartición de variables entre esquemas. En el ejemplo de la figura, tenemos dos esquemas, *schema1.c* y *schema2.c*. El primero de ellos, exporta una variable. Finalmente *schema2.c* importa la variable que *schema1.c* ha exportado.

1. schema1.c realiza una llamada a una función de `schemas.c`; `putSchemaVar()`. Le pasa como parámetros toda la información necesaria para realizar la exportación correctamente.
2. `schemas.c` inserta un nuevo registro en el censo de variables que tenemos en jde.c.
3. Posteriormente `schema2.c` desea importar el símbolo que `schema1.c` exportó previamente. Realiza por tanto una llamada a `getSchemaVar()` para obtener el puntero a la variable

4. `schemas.c` consulta la estructura de datos de `jde.c` para localizar la variable en el censo

5. Posteriormente se localiza la variable, por lo que `schemas.c` ya tiene localizado el puntero a la variable

6. Finalmente `schemas.c` devuelve a `schema2.c` el puntero a la variable que deseaba importar.

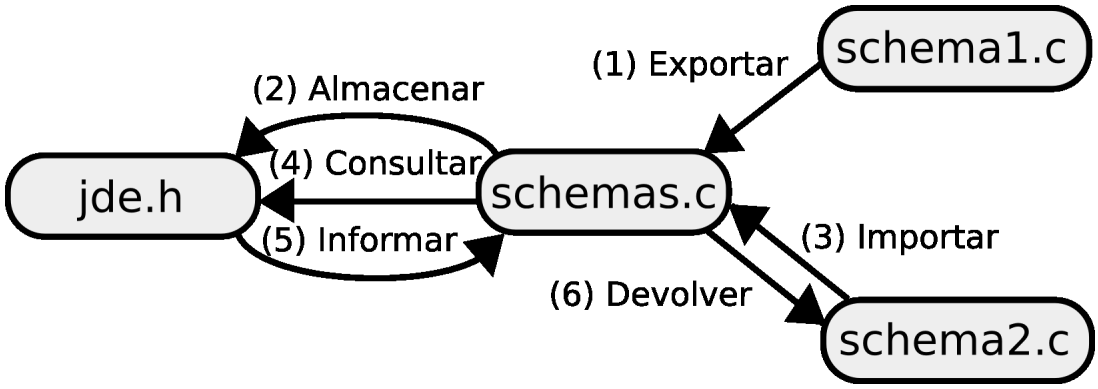


Figura 4.2: Proceso de importación/exportación de variables

4.2.4. Jerarquía de esquemas

Si bien la jerarquía se comporta del mismo modo que en `jde.c`, la infraestructura se ha reimplementado por completo, haciendo uso de los mecanismos proporcionados por `schemas.c`.

Como vimos en la sección 1.3, los esquemas pueden organizarse jerárquicamente de forma dinámica, de modo que el conjunto de hijos de un determinado padre, realiza un bloque de trabajo que finalmente es manejado y administrado por el padre. En determinados casos no habrá ningún problema en la ejecución concurrente de determinados esquemas. Sin embargo hay casos en los que se hace necesario un arbitraje para decidir qué esquema entra en funcionamiento. La diferencia entre un comportamiento y otro viene determinada por el tipo de esquema que estemos utilizando. Tenemos dos tipos de esquemas, perceptivos y de actuación.

Esquemas perceptivos

Los perceptivos son aquellos que no modifican ningún recurso ajeno a él. Pueden por tanto leer variables de la API de `jdeneo.c` o de otros esquemas, pero sin modificar ningún elemento que esté fuera de su espacio de nombres. Este tipo de esquemas tan solo tiene dos estados, por lo que o bien están *dormidos*, o bien están en *ejecución*. El paso de uno a otro es automático (Ver figura 4.3) y se produce realizando llamadas a `schema_startup` y `schema_suspend`.

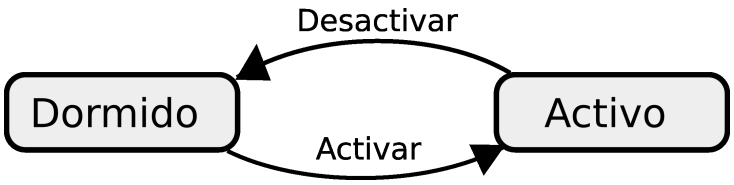


Figura 4.3: Diagrama del funcionamiento de un esquema perceptivo

Esquemas de actuación

En `jdeneo.c` se crea el concepto de grupo, de modo que no todos los esquemas de actuación, hijos de un determinado esquema, compiten necesariamente entre sí por el control (Ver sección 1.3.1). Esto se debe a que existen casos en los que la ejecución de un conjunto de esquemas no interfiere la ejecución de otro conjunto, por lo que pertenecen a grupos distintos y no reciben el mismo arbitraje.

Los esquemas de actuación, a diferencia de los perceptivos, son aquellos que van a modificar el estado de algún recurso ajeno a él. Pueden leer y escribir en variables tanto de `jdeneo.c` como de otros esquemas. Los padres tendrán uno o más grupos de hijos de este tipo. En este contexto surge el problema de varios esquemas de un mismo grupo tratando de acceder simultáneamente a un mismo recurso para su modificación. Esta situación es inadmisible, y debe ser el padre el que decida quién debe ejecutarse por medio de la función de arbitraje. Este tipo de esquemas se caracteriza también, como ya hemos visto, por tener una serie de precondiciones tales que si se cumplen, entonces el esquema es candidato a estar activo.

Los esquemas de actuación tienen cuatro estados asociados:

1. *dormido* – El esquema no está activado. No consume CPU.
2. *alerta* – No se satisfacen las precondiciones y comprueba si se cumplen.
3. *preparado* – Se cumplen las precondiciones del esquema pero no está activo.
4. *activo* – Se está ejecutando.

El paso de estado *dormido* a estado *alerta* viene determinado por el padre, que preactiva al esquema. En el caso de que no se cumplieran las precondiciones del esquema, se chequearía si alguno de sus hermanos está en *preparado* o en ejecución, de modo que si no hubiera ninguno (Vacío de control) se llamaría a la función de arbitraje para que el padre decida qué acción realizar. Una vez que se cumplan las precondiciones se pasará automáticamente a estado *preparado*. En este momento el esquema es candidato a ser *activo*, por lo que se comprueba si alguno de sus hermanos lo está. Si no hubiera ninguno, entonces pasaría automáticamente a estado *activo* y se procedería con la ejecución principal. Sin embargo, si alguno de sus hermanos ya estuviera en estado *activo* (Solape de control), se llamaría a la función de arbitraje para que el padre decida quién debe continuar la ejecución.

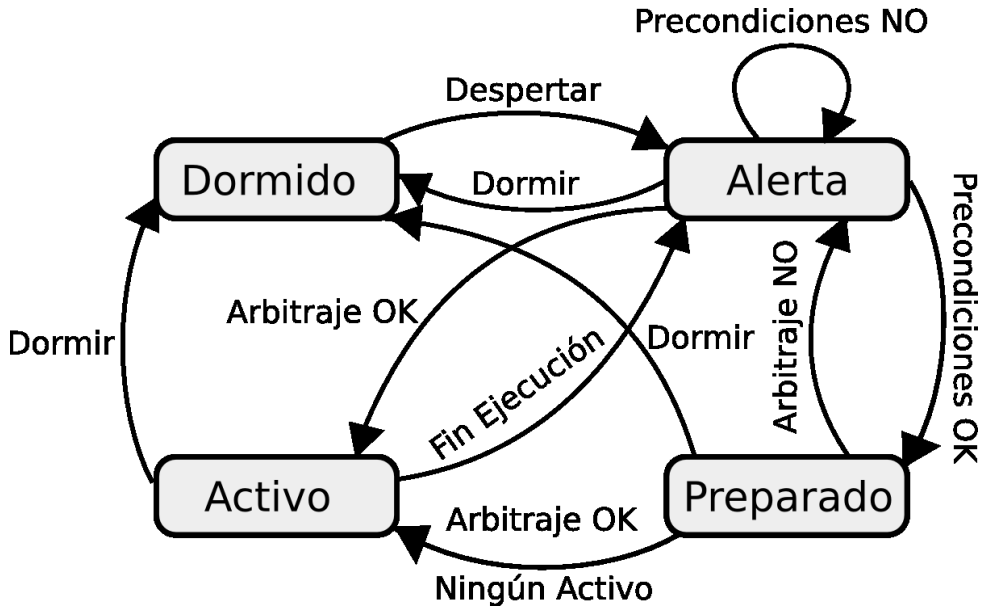


Figura 4.4: Diagrama del funcionamiento de un esquema de actuación

Para conocer si algún esquema de su grupo está en estado *activo*, y detectar solapes de control, se realiza una llamada a la función de `jdeneo.c schemaIsAnyWinner()`, que realiza las comprobaciones necesarias en el censo. Del mismo modo para conocer si hay algún esquema *preparado*, y detectar vacíos de control, se realizarán llamadas a la función `schemaIsAnyReady()`.

```
gboolean schemaIsAnyWinner(gchar *id);
gboolean schemaIsAnyReady(gchar *id);
```

La función de arbitraje es básicamente un procedimiento que comprueba el estado actual del grupo de hijos, y de acuerdo a la información obtenida decide quién debe ejecutarse. Se emplea como hemos visto en dos situaciones: (1) En caso de vacío de control, cuando hay esquemas preactivados y no se cumplen las precondiciones de ninguno, y (2) en caso de solape de control, cuando se cumplen las precondiciones de más de un esquema. Se ha implementado con la siguiente interfaz:

```
gboolean schema_arbitration(gchar *id);
```

Recibe un sólo argumento, que es el identificador del hijo. De este modo, cuando un esquema detecte una situación crítica, informará al padre y éste le dirá si él debe ejecutarse o no.

Las precondiciones se basan en estímulos externos e internos, gracias a los cuales se determina si el esquema se encuentra en situación favorable para la ejecución o no. La implementación de la función que comprueba las precondiciones es un procedimiento que devuelve un booleano informando del resultado.

```
gboolean precondiciones();
```

4.3. Visualización 3D

Como vimos en 2.1.1, uno de los puntos flacos de `jde.c` es el visor en 2D de la escena reconocida por el robot. La herramienta escogida para el modelado en 3D en *jdeneo.c* es OpenGL. Además se buscaba un *renderizado* del robot lo más realista posible, de modo que fuera posible visualizar cada parte en funcionamiento.

La herramienta utilizada para asistir en el modelado del robot Pioneer ha sido Blender. El trabajo consistió en fotografiar detalladamente el robot y tomar medidas, para diseñarlo posteriormente con Blender[5] (Ver figura 4.5). Finalmente, exportamos el robot a un fichero de código fuente en C en el que cada una de las funciones contiene el código OpenGL para *renderizar* un objeto.

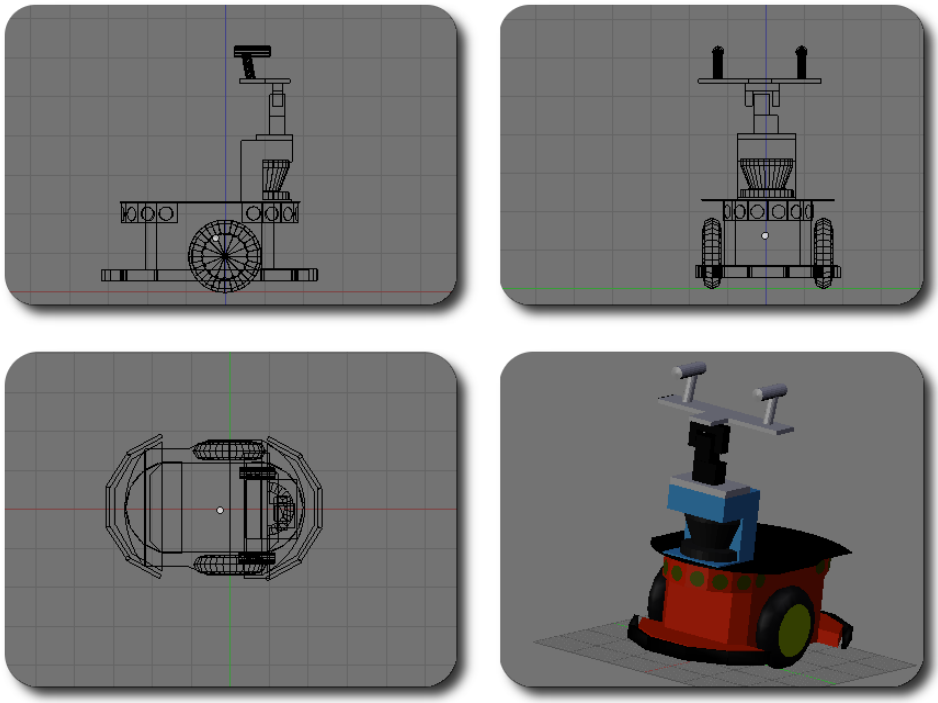


Figura 4.5: Modelado del robot Pioneer.

Para visualizar la escena final reconocida por el robot se deben cumplir ciertos requisitos:

1. Debe ajustarse al tamaño de la ventana, haciendo uso de todo el espacio posible.
2. Debe *renderizarse* frecuentemente, de modo que haya sensación de movimiento continuo.

Para la programación de la visualización de la escena hemos recurrido al uso de señales GDK y de funciones *timeout*. Podemos distinguir dos grupos de funciones en la visualización 3D: por un lado tenemos aquellas que son llamadas a partir de estímulos externos (señales GDK), y por otro lado aquellas que se ejecutan de forma repetida en forma de *timeouts*.

[5]Blender es un software libre multiplataforma dedicado especialmente al modelado y creación de gráficos tridimensionales

Inicialización y configuración

Entendemos por inicialización a aquel evento que sucede sólo una vez a lo largo del ciclo de vida de la escena 3D. En este contexto es en el que estableceremos los parámetros globales de la escena. Principalmente podemos distinguir entre las siguientes acciones:

- Asignación del foco de luz. Establecemos las coordenadas, intensidad y difusión de la fuente de luz global de la escena.
- Punto de vista. Inicialmente se situará el punto de vista del observador en unas determinadas coordenadas
- Parámetros OpenGL. Finalmente se establecen los flags necesarios para el *renderizado* de la escena, y se carga la matriz identidad para la proyección.

Debido a que el código de inicialización sólo debe ejecutarse una vez tras la creación del objeto GDK en el que proyectaremos la escena, hemos asociado la señal *realize* de dicho objeto a la función de inicialización.

```
g_signal_connect_after (G_OBJECT (drawing_area), "realize",
   G_CALLBACK (realize), NULL);
```

Además de la inicialización, tendremos también la configuración, o modulación, que es la modificación en tiempo de ejecución de los parámetros de OpenGL. Esta funcionalidad debe ejecutarse en situaciones en las que se altere el estado del objeto contenedor de la escena, como por ejemplo en el redimensionado de la ventana. En este caso procedemos del mismo modo que con la función de inicialización, y atamos el evento *configure_event* a la función que realiza el tratamiento necesario.

```
g_signal_connect (G_OBJECT (drawing_area), "configure_event",
   G_CALLBACK (configure_event), NULL);
```

Pese a que tengamos controladas la mayoría de las situaciones que afectan al contexto del objeto en el que proyectamos la escena, hay casos que necesitan una funcionalidad más. Hablamos de las situaciones en que, sin modificar necesariamente el estado del objeto, alteramos la región de pantalla que está empleando. Ésto sucede, por ejemplo, cuando deslizamos el cursor del ratón sobre la escena, o cuando temporalmente ocultamos parte o la totalidad de la imagen. Para ello ataremos una tercera función a *expose_event*, el evento encargado de detectar este tipo de situaciones:

```
g_signal_connect (G_OBJECT (drawing_area), "expose_event",
   G_CALLBACK (expose_event), NULL);
```

La función expose_event es la encargada de realizar el *renderizado* de todos los objetos. En el siguiente apartado se describe con más detalle.

Animación

En el cine se logra el efecto de animación proyectando 24 imágenes por segundo en la pantalla. Con este método se logra percibir sensación de movimiento. Para lograr este efecto debemos tener un procedimiento encargado de refrescar la escena. Debido a que la finalidad de la visualización 3D de la escena no es una tarea relevante para el correcto funcionamiento del robot, no se hace necesario un control en tiempo real. Hemos considerado que el mecanismo adecuado para esta situación son las funciones *idle*[6] de GLib. De este modo, la escena se refresca frecuentemente y, por ejemplo, se verán reflejados de forma automática los movimientos del robot.

```
idle_id = g_idle_add_full (GDK_PRIORITY_REDRAW,
                           (GSourceFunc) idle,
                           drawing_area,
                           NULL);
```

La función con la que hemos asociado al *idle* es `expose_event()`. Esta función, como ya hemos visto, es la que procesa la información proveniente de la API de variables del núcleo de `jdeneo.c`. Gracias al fichero generado por Blender, tenemos una biblioteca que nos provee de una función para *renderizar* cada uno de los objetos que compone el robot. La función `expose_event` es la encargada de realizar las llamadas a las funciones de *renderizado* de objetos.

La *renderización* del robot se realiza sobre el sistema de referencia absoluto. Sin embargo hay objetos que tienen su propio sistema de coordenadas. Por ejemplo, las cámaras dependen del cuello mecánico, por lo que su sistema de coordenadas es solidario al cuello mecánico. La solución algebraica al cambio de base se realiza mediante multiplicación de matrices. Esta tarea es muy común en el *renderizado* de escenas 3D, motivo por el cual OpenGL se encarga de realizarlo de forma sencilla haciendo uso de la GPU, liberando de ese modo al procesador principal. Además, nos permite añadir y quitar matrices de transformación de un modo sencillo e intuitivo, mediante pilas.

Toda esta funcionalidad se debe añadir al código de la función `expose_event()` para poder *renderizar* cada objeto por separado y de acuerdo a su propio sistema de coordenadas. De ese modo, una porción del código de `expose_event()` es:

```
cargarBaseGiroCuello(); /* Renderizamos base del giro del cuello */
glPushMatrix();         /* Creamos un nuevo contexto */
   glTranslatef(-0.1f, -1.8f, 5.0f);  /* Lo llevamos a su sitio */
   glRotatef(15.0, 1.0f, 0.0f, 0.0f); /* Lo giramos */
   glTranslatef(0.1f, 1.8f, -5.0f);   /* Lo llevamos al origen */
   cargarCamaras();     /* Renderizamos camaras */
   cargarBaseCamaras(); /* Renderizamos base de las cámaras */
```

[6] Una función *idle* es aquella que se ejecuta cuando no hay eventos más prioritarios pendientes de ejecución.

```
   cargarGiroCuello();  /* Renderizamos giro del cuello */
glPopMatrix();          /* Retornamos al contexto anterior */
```

En la figura 4.6(a) podemos ver el resultado de la *renderización* utilizada en Blender. Observamos que se obtiene un resultado muy rico en matices. Esto se debe al motor de trazado de rayos empleado por Blender, YafRay.

Sin embargo, si observamos el resultado de una captura del mismo robot en *jdeneo.c* (Ver figura 4.6(b)) podemos comprobar cómo se han perdido matices y definición. Ésto se debe sobretodo a dos motivos: Por un lado, en el proceso de exportación de Blender a código C se pierde mucha información, y por otro lado, el *renderizado* en jdeneo.c se realiza en tiempo real sin emplear motores de trazado de rayos.

(a) *Renderizado* en Blender (b) *Renderizado* en jdeneo.c

Figura 4.6: *Renderizados* del robot Pioneer

4.4. Interfaz Gráfica

El modo de acceso del usuario humano a los sensores y actuadores que provee la plataforma `jdeneo.c` es la interfaz gráfica. En concreto permite teleoperar los actuadores, comprobar el correcto funcionamiento de los sensores, activar y desactivar esquemas, visualizar la escena 3D[7], acceder a la interfaz gráfica de los esquemas, etc. Utilizaremos la herramienta de diseño de interfaces Glade, y programaremos todos los manejadores asociados a cada señal.

4.4.1. Disposición de los elementos

Como ya hemos dicho, hemos buscado una interfaz elegante e intuitiva, planteando como principal objetivo la usabilidad. A grandes rasgos, la interfaz gráfica de la infraestructura se divide en dos secciones bien diferenciadas entre sí. Por un lado tenemos la barra de herramientas y por otro el espacio de usuario organizado por pestañas. Desde la barra de herramientas tendremos acceso a acciones globales, como la apertura de esquemas, o la salida de la aplicación. En el espacio de usuario tendremos tres pestañas en las que se muestra con toda la información

[7]Si bien uno de los componentes lógicos de la interfaz gráfica es la visualización 3D, hemos considerado que al ser un objetivo con entidad propia sería preferible dedicarle su propio apartado: Sección 4.3

relevante para el usuario y se le permite interactuar con el robot.

A la hora de dividir en pestañas hemos realizado distinción entre tres bloques, independientes entre sí. Por un lado tenemos el panel desde el que se manejan e informa de los valores de los sensores y actuadores del robot, por otro la visualización 3D de la escena reconocida por el robot, y por último el control y visualización de esquemas.

Por otra parte, como ya hemos visto, los esquemas tienen su propia interfaz gráfica. Ésta se muestra en una nueva ventana. De este modo es posible visualizar a la vez la interfaz de la infraestructura y la de los esquemas.

Interfaz de sensores y actuadores

Al igual que su predecesor, *jdeneo.c* provee de mecanismos que permiten al usuario humano interactuar desde la interfaz gráfica con el robot (Ver figura 4.7). Gracias a este soporte, es posible teleoperar el robot, o comprobar visualmente los estímulos obtenidos desde los sensores, sin necesidad de cargar ningún esquema. La principal funcionalidad que aporta todo esto es la depuración en tiempo real.

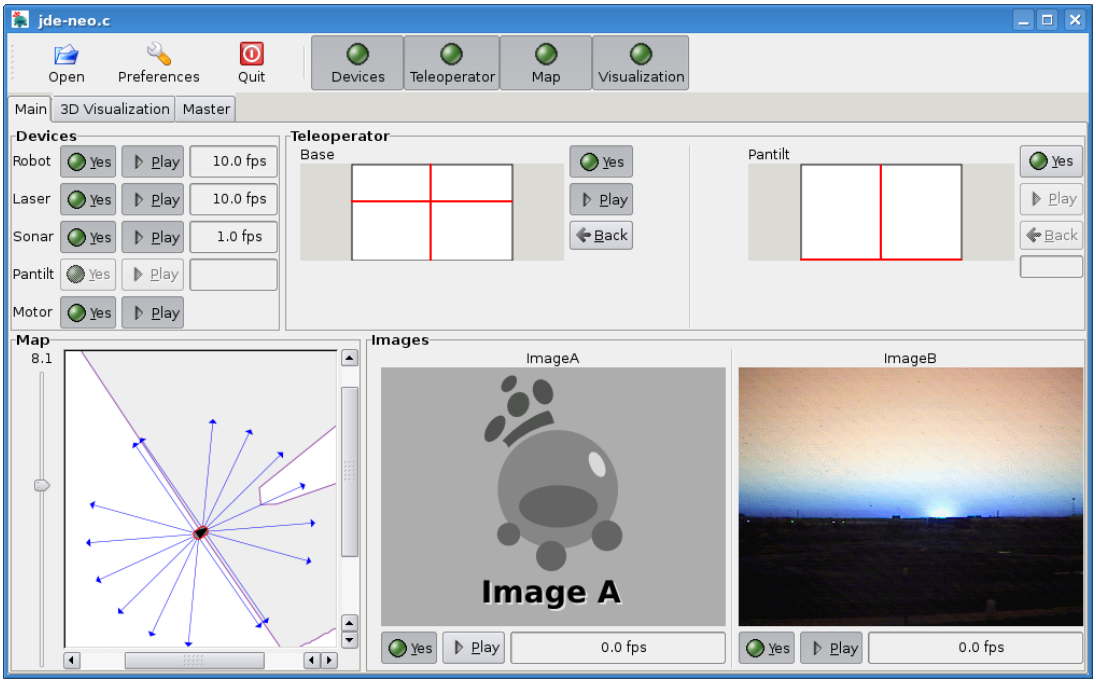


Figura 4.7: Interfaz de sensores y actuadores en funcionamiento.

Hemos dividido en marcos las funcionalidades que provee la interfaz de sensores y actuadores:

- Dispositivos (Hardware de referencia). En este marco se encuentra la interfaz de activación y ejecución de la mayoría de los dispositivos del robot. En concreto tenemos (1) odometría del robot, (2) sensores láser, (3) sensores sónar, (4) motor del robot y (5) cuello mecánico. Se muestra también el número de iteraciones por segundo de cada uno de los dispositivos activos.

- Teleoperadores. El modelo de robot Pioneer con el que trabajamos presenta dos actuadores: cuello mecánico y motores. Desde este marco podremos activarlos y usarlos a través de sendos posicionadores.
- Mapa 2D. Uno de los métodos que proporciona la plataforma para la visualización de los estímulos del robot es el mapa. Empleando un *canvas*, observamos el movimiento del robot y los resultados de los sensores láser y sónar.
- Imágenes. Empleado para poder comprobar qué imagen está recibiendo el robot por las cámaras.

Interfaz de visualización 3D

`jdeneo.c` contiene visualización tridimensional de la escena reconocida por el robot de forma nativa. Para ello hemos creado una interfaz mínima en la que se muestra la escena observada desde una cámara virtual. Para seleccionar los elementos que se desean visualizar hemos añadido un menú contextual, de ese modo el usuario puede escoger si visualizar o no el suelo, los ejes de coordenadas del sistema absoluto, los sensores láser y los sensores sónar.

El comportamiento de la cámara virtual se ha simplificado todo lo posible:

- La cámara siempre apunta al robot.
- Para alejarnos o acercanos de la escena usaremos el *scroll* del ratón.
- Para cambiar las coordenadas de la cámara haremos *click* en la escena y arrastraremos hasta llegar a la posición deseada.

Podemos ver la visualización 3D en funcionamiento en la figura 4.8.

Interfaz de control de esquemas

Para manejar y visualizar el estado de los esquemas hemos creado una tercera pestaña (Ver figura 4.9). En este marco se muestra la siguiente información para cada esquema cargado:

- Identificador del esquema. Será el nombre especificado en `schema_id` (Ver 4.2.2).
- Estado. Usaremos un convenio de colores para marcar el estado del esquema
  1. Negro – Esquema en estado *dormido*.
  2. Rojo – Esquema en estado *alerta*.
  3. Naranja – Esquema en estado *preparado*.
  4. Verde – Esquema en estado *activo*.
- Visualizar. *Checkbox* desde el que se puede activar y desactivar la interfaz gráfica de un esquema, realizando llamadas a `schema_gui_startup` y `schema_gui_suspend` respectivamente.

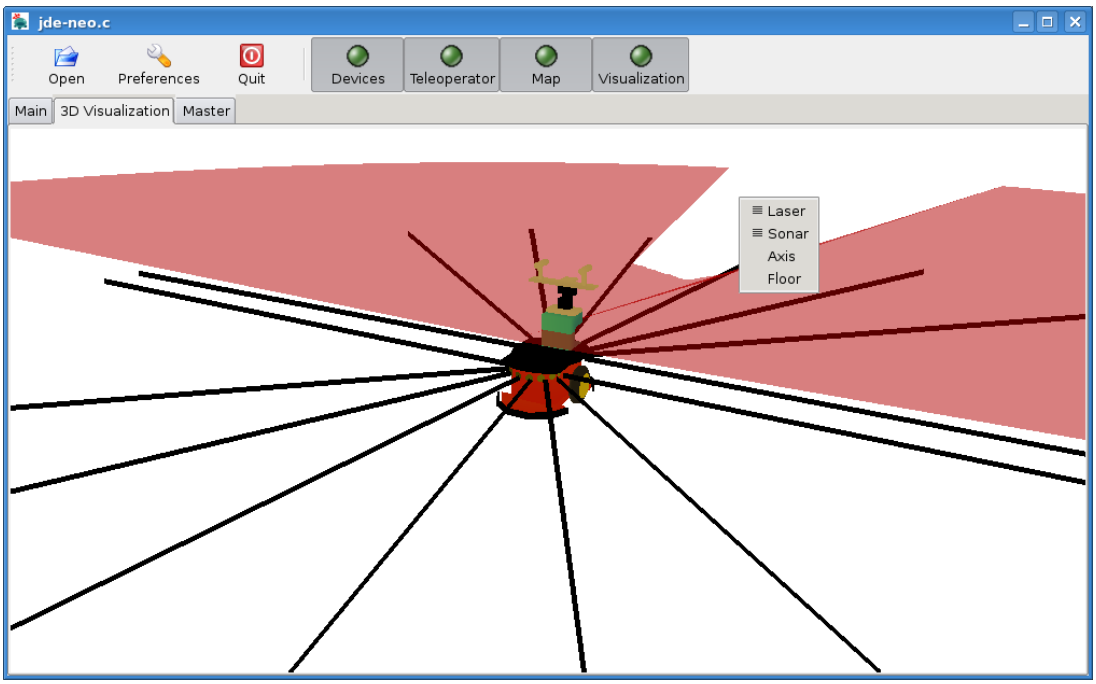


Figura 4.8: Interfaz de visualización 3D.

- Activar. *Checkbox* para iniciar o detener la ejecución de un esquema, realizando llamadas a `schema_startup` y `schema_suspend` respectivamente.
- Tiempo de ciclo. Campo de texto para visualizar y modificar el valor de `schema_interval` (Ver 4.2.2).

4.4.2. Programación de la interfaz gráfica

La programación de la interfaz gráfica ha sido significativamente minimizada debido al uso del diseñador Glade (Ver 3.1.1). Sin embargo, la mayor parte de la funcionalidad se ha materializado en forma de código fuente. Atendiendo a los mecanismos empleados, podemos clasificar estas funcionalidades en dos tipos: Hebras de control y *callbacks*.

Hebra principal y hebras secundarias

La interfaz gráfica es la vía principal de comunicación entre el usuario humano y la plataforma. Debido a eso se hacen necesarios una serie de requisitos:

1. La interfaz debe funcionar como una hebra independiente del resto de la plataforma, de modo que se ejecute concurrentemente.
2. Se debe inicializar la interfaz desde el código de inicialización de la plataforma, esto es, desde la función *main*.

```
int main(int argc, char** argv) {

   /* Inicialización de la aplicación */
```

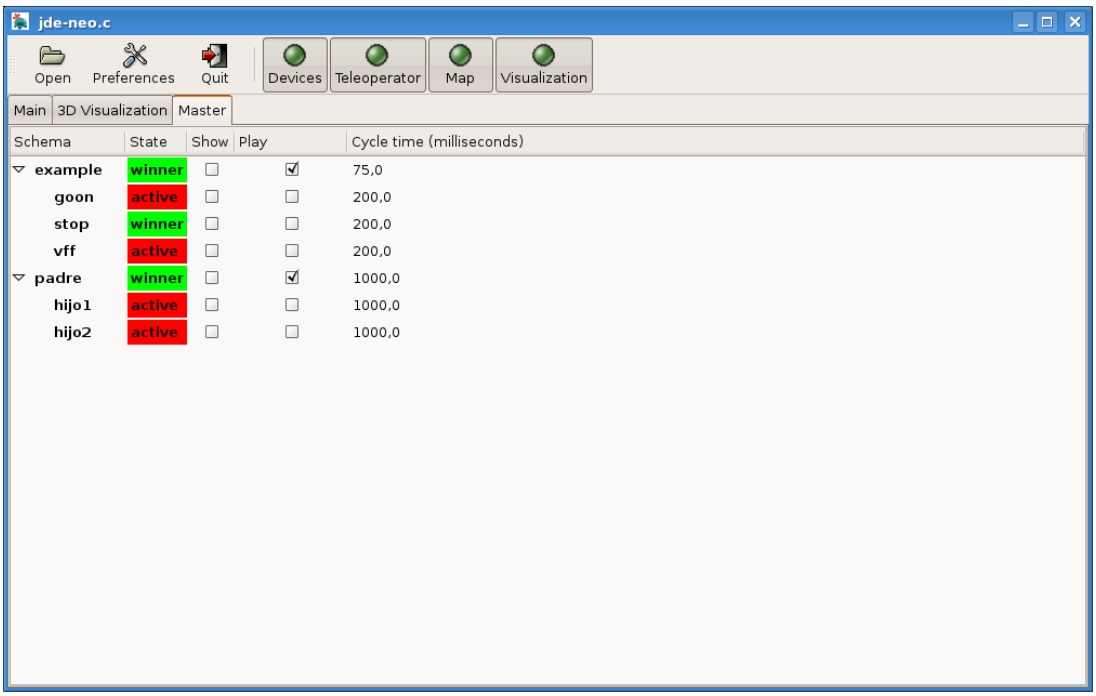


Figura 4.9: Interfaz de control de esquemas.

```
   if (!g_thread_create(guiStartup, NULL, FALSE, &error)) {
      g_printerr ("Failed to initialize gui thread: %s\n",
         error->message);
      return 1;
   }

   /* Resto de instrucciones */

}
```

La función encargada de inicializar toda la interfaz gráfica es, como vemos, guiStartup. Su tarea consiste en lo siguiente:

1. Inicializar variables globales a toda la interfaz gráfica.

2. Cargar visor de esquemas. El visor de esquemas utiliza una estructura GtkTreeView que no es posible definir desde Glade. Debido a eso se hace necesaria una función que cree el objeto en tiempo de ejecución.

3. Cargar esquemas por defecto. En las preferencias de jdeneo.c existe una variable en la que asignamos una lista de directorios donde queremos que se busquen esquemas. Es en este momento en el que se cargan todos aquellos que se encuentren.

4. Inicializar visor 3D. Al igual que con GtkTreeView, el visor de la escena tridimensional se ha tenido que crear con código, de modo que se instancia en tiempo de ejecución.

5. Crear bucle principal. Una vez que se ha inicializado el código principal, ya se está en condiciones de entrar al bucle principal de gtk. Este bucle será el encargado de recibir eventos, informar mediante señales, y realizar todas las tareas necesarias, tal y como vimos en 3.1.

Además de la hebra principal, hay otras hebras que se crean a lo largo del ciclo de vida de la aplicación. En concreto tenemos dos: La primera de ellas se encarga de actualizar el valor de las iteraciones por segundo de cada dispositivo. La segunda tan solo se encarga de mostrar una pantalla de bienvenida (*Splash*) y de destruirla cuando se cumple un determinado evento.

Manejadores de señales – *Callbacks*

Como vimos en la sección 3.1, el mecanismo de eventos GTK es tal que asociamos señales a manejadores. El diseñador de interfaces Glade permite asociar las señales de un determinado *widget* a sus funciones manejadoras o *callbacks*.

Este tipo de funciones reciben siempre como parámetro una referencia al *widget* del que procede la señal, pudiendo reconocer la fuente del evento. De ese modo, hemos podido emplear en algunos casos un mismo *callback* para manejar eventos de distintos objetos.

A continuación mostramos a modo de ejemplo el código del *callback* que maneja el evento "toggled" del botón que activa el motor del robot:

```
void on_motor_yes_toggled(GtkToggleButton *togg, gpointer user_data) {
   if (gtk_toggle_button_get_active(togg)) {
      motors_resume(NULL, NULL);
      gtk_widget_set_sensitive(glade_xml_get_widget(xml, "motor_play"), TRUE);
   }
   else {
      motors_suspend();
      gtk_widget_set_sensitive(glade_xml_get_widget(xml, "motor_play"), FALSE);
   }
}
```

Capítulo 5
Resultados experimentales

Una vez descrita la implementación propuesta para resolver los objetivos fijados en el capítulo 2, veremos en profundidad algunos resultados experimentales sobre la plataforma desarrollada. En primer lugar, comprobaremos cómo se ha procedido a la construcción tanto de la aplicación principal, como de esquemas sencillos. Posteriormente mostraremos esquemas básicos que ponen de manifiesto las principales funcionalidades facilitadas por `jdeneo.c`. Finalmente veremos un ejemplo real, en el que se implementará un algoritmo de navegación segura.

5.1. Construcción del sistema y esquemas

El proceso de construcción o compilación es aquel que traduce el código fuente a código objeto. Apoyándonos en bibliotecas externas veremos cómo construir e instalar la plataforma desarrollada, y cómo construir los esquemas en forma de bibliotecas dinámicas.

5.1.1. Construcción del sistema

`jdeneo.c` está desarrollado en el lenguaje de programación de alto nivel `C`. Emplea para la mayoría de operaciones de bajo nivel las rutinas proporcionadas por la biblioteca de propósito general `GLib` (Ver sección 3.2). Sin embargo, los `drivers` heredados de `jde.c` se han mantenido en su mayor parte intactos, motivo por el cual `jdeneo.c` requiere el uso de otras bibliotecas.

A continuación se enumeran todas las bibliotecas no estándar que son necesarias tener instaladas para la correcta construcción del sistema: A lo largo del desarrollo de `jdeneo.c` hemos empleado `libglib2.0` como bibliotecas de bajo nivel de propósito general; Para poder realizar la carga de la interfaz gráfica en tiempo de ejecución utilizamos `libglade2.0`; Como ya vimos, necesitamos también una biblioteca auxiliar para incluir un *canvas* en la aplicación, en concreto `libgnomecanvas2.0` del proyecto Gnome; Otra de las bibliotecas auxiliares que hemos utilizado ha sido `libgtkglext1`, que nos facilita objetos y métodos para empotrar visualización OpenGL a `jdeneo.c`; Y por último, para poder materializar el acceso al bus IEEE1394, *Firewire*, y manejar cámaras en él, hemos usado `libgraw1394-8` y `libdc1394-13` respectivamente. Además de las bibliotecas que acabamos de mencionar, emplearemos también otras bibliotecas estándar en entornos `POSIX`, como por ejemplo `pthread`.

En entornos UNIX para facilitar el proceso de compilación existe la utilidad `make`, que a partir de un fichero `makefile`, en el que se incluyen las directrices a seguir, construye la aplicación.

A tal efecto hemos escrito un sencillo fichero `makefile` para automatizar la tarea de generación del sistema. La única precondición es, como vimos en el apartado anterior, tener instaladas las bibliotecas necesarias.

`make` es el encargado de invocar al compilador y de proporcionarle todos los parámetros necesarios. Debido a eso, es necesario que en el fichero `makefile` estén especificadas, explícita o implícitamente, las rutas a las bibliotecas y a los ficheros de cabeceras necesarios para la construcción. La solución adoptada resuelve dinámicamente las rutas, gracias a la aplicación `pkg-config`, que genera las rutas absolutas a las bibliotecas o a las cabeceras de las bibliotecas que se le especifiquen.

En nuestro `makefile` todas las rutas vienen descritas por `pkg-config`:

```
INC-DIR = ‘pkg-config --cflags libglade-2.0 gthread-2.0 libgnomecanvas-2.0 \
   gtkglext-1.0 gdkglext-1.0‘
LIB-DIR = ‘pkg-config --libs libglade-2.0 gthread-2.0 libgnomecanvas-2.0 \
   gtkglext-1.0 gdkglext-1.0‘
```

Además, es necesario un argumento en el que especificamos la ruta en la que se localizarán lo recursos necesarios por la aplicación. Estos recursos son las imágenes utilizadas por la interfaz, los iconos, y el fichero .xml que define la interfaz gráfica. Esta ruta la proporcionamos en el fichero `makefile` por medio de la macro `RESOURCEDIR`.

Finalmente, para construir el sistema tan sólo debemos ejecutar `make` en el directorio raíz del código fuente de la aplicación.

5.1.2. Construcción de un esquema

Mediante el proceso de construcción de esquemas generaremos, a partir del código fuente del mismo, una biblioteca dinámica. De ese modo, el núcleo de la aplicación podrá cargarlo en tiempo de ejecución, solucionando por tanto los problemas acarreados por el enlazado estático.

Del mismo modo que para compilar el sistema se hace necesario el enlazado con un conjunto de bibliotecas, los esquemas también tendrán unas bibliotecas necesarias para poder construirse. Ésto se debe a que es condición necesaria enlazar los esquemas con el fichero de cabeceras `jde.h`[1], y por consiguiente, con todas las bibliotecas empleadas en él.

Gracias a la utilidad `pkg-config` será muy sencillo incluir todas las rutas pertinentes para el enlazado en tiempo de compilación. Veámoslo con un ejemplo:

```
gcc -g -Wall -c ‘pkg-config --cflags gmodule-2.0 libglade-2.0‘ schema1.c
gcc -g -Wall ‘pkg-config --libs gmodule-2.0 libglade-2.0‘ -shared -o \
   schema1.jde schema1.o
```

[1] De ese modo pueden acceder a la API de variables y funciones que la arquitectura provee a los esquemas.

A continuación explicaremos otros requisitos en el proceso de construcción de esquemas estudiando el ejemplo de compilación propuesto. La utilidad `pkg-config` nos genera de forma automática todas las rutas necesarias a partir de las dos bibliotecas que le ofrecemos como argumento, de ese modo ya están solucionados los *paths*. Es importante destacar que todos los esquemas deben incluir necesariamente enlazado a las bibliotecas gráficas `libglade`, independientemente de si el esquema concreto tendrá o no interfaz gráfica.

Por otro lado observamos el argumento `-shared`. Gracias a este parámetro construimos el esquema como biblioteca dinámica. Vemos, finalmente, que el fichero de salida tiene el sufijo `.jde`: `-o schema1.jde`. Empleando este convenio la aplicación principal localizará fácilmente las bibliotecas dinámicas que están construidas para su enlazado con `jdeneo.c`.

5.2. Pruebas de mecanismos

En esta sección comprobaremos de forma experimental el correcto funcionamiento de la plataforma. Trataremos de demostrar cada una de las principales funcionalidades que ofrece la arquitectura desarrollada.

5.2.1. Comunicación entre esquemas

Los esquemas se comunican con la arquitectura a través de una API preestablecida (Ver sección 4.2.2). Sin embargo, deben poder "personalizar" sus símbolos exportados para poder proveer modulación y variables sensoriales depuradas. A tal efecto, como vimos en el apartado 4.2.3, hemos desarrollado el censo de variables, en el que quedan registrados todos los símbolos que exportan los esquemas.

En el ejemplo propuesto vamos a comprobar cómo y cuándo debe un esquema tanto exportar como importar los símbolos. Además, para constatar el correcto comportamiento del sistema, los identificadores de los símbolos exportados serán los mismos. De ese modo, tendremos dos esquemas: `schema_export1` y `schema_export2`. Los dos exportarán una variable de tipo entero llamada `shared`.

Cómo exportar e importar símbolos

Como vimos en el apartado 4.2.3 tenemos dos funciones que nos permiten exportar e importar variables. Son las encargadas de acceder al censo de variables para realizar las tareas pertinentes.

```
void *getSchemaVar(gchar *id, gchar *varname);


void putSchemaVar(gpointer var, gchar *varname, gchar *schema);
```

Cuándo y dónde exportar símbolos

Si bien es posible exportar los símbolos en cualquier momento, una configuración típica debería exportarlos en la primera ejecución de `schema_startup()`. Así, estarán todos sus símbolos

disponibles desde que el esquema es cargado por la plataforma, y se evitan casos en que otro esquema desea importar una variable antes de que ésta haya sido publicada.

```
int shared = -2;

 ...

G_MODULE_EXPORT void schema_startup (void) {
   if (loaded) {
      ...
   }
   else {
      loaded = TRUE;
      g_print ("\n--- Startup de esquema %s ---\n", schema_id);

      /* La primera vez exporta sus símbolos */
      putSchemaVar((gpointer)&shared, "shared", schema_id);
   }
}
```

En este ejemplo, el proceso de exportado de sus respectivas variables `shared` ha sido realizado tanto por `schema_export1` como por `schema_export2`.

Cuándo y dónde importar símbolos

Del mismo modo que exportamos los símbolos en tiempo de carga, la resolución de nombres mediante el proceso de importado se realizará en la activación de ejecución de los esquemas, es decir, en las sucesivas llamadas a `schema_startup()`. Veamos cómo obtiene y almacena `schema_export1` el símbolo `shared` de `schema_export2`:

```
int *shared_externo;
...

G_MODULE_EXPORT void schema_startup (void) {
   GError *error;

   if (loaded) {
      /* Si no es la primera vez, se inicia la hebra de schema_iteration */
      schema_thread = gtk_timeout_add(schema_interval,
         (GtkFunction)schema_iteration, &error);
      if (!schema_thread) {
         g_printerr ("Failed on schema_startup: %s\n", error->message);
         schemaSetState(schema_id, slept);
      }
```

```
      else {
         schemaSetState(schema_id, winner);
         shared_externo = getSchemaVar("schema_export2", "shared");
      }
   }
   else {
      ...
   }
}
```

Hemos almacenado en la variable `shared_externo` de `schema_export1` un puntero al símbolo `shared` de `schema_export2`.

Finalmente, para comprobar el correcto comportamiento, se muestra por pantalla mediante la función `schema_iteration()` el valor de la variable importada en `schema_export1`:

```
gboolean schema_iteration(gpointer data) {
   g_print("Variable externa = %d\n", *shared_externo);
   return TRUE;
}
```

El valor mostrado es, efectivamente, el esperado. Podemos concluir que el proceso de importado/exportado de variables se comporta correctamente.

5.2.2. Jerarquía de esquemas

La arquitectura cognitiva JDE establece que los esquemas pueden ordenarse por niveles formando jerarquías. Gracias a ello, varios esquemas pueden trabajar en conjunto, gobernados por un padre, para lograr un objetivo concreto.

En `jdeneo.c`, al igual que en `jde.c`, se ha realizado un diseño en el que la mayoría de los mecanismos necesarios para llevar a cabo la jerarquía están implementados dentro de los propios esquemas. Por ejemplo, la "máquina de estados" de un determinado hijo (Ver figura 4.4) está codificada directamente en el código de ese esquema.

El ejemplo que describimos a continuación contiene tres esquemas. El primero de ellos, llamado `padre`, será el que gobierne la jerarquía. Los dos restantes, `hijo1` e `hijo2`, serán esquemas de actuación mutuamente excluyentes: No pueden funcionar a la vez.

El comportamiento será el siguiente: Si el valor de `laser[90]` está entre 4500 y 8000 milímetros entonces `hijo1` puede ejecutarse activando una velocidad de 200. Del mismo modo, si el valor de `laser[90]` se encuentra entre 1000 y 5000 milímetros, `hijo2` puede ejecutarse activando una velocidad de 100. Existe posibilidad de solape de control cuando la medidas del laser se encuentren entre 4500 y 5000 milímetros. También hay vacío de control cuando el resultado del

laser se encuentre por encima de 8000 o por debajo de 1000 milímetros. En esos casos debe ser el padre quien decida cómo resolver el conflicto mediante la función de arbitraje. Podemos ver el diagrama 5.1 para comprobar de forma visual cuáles son las secciones críticas.

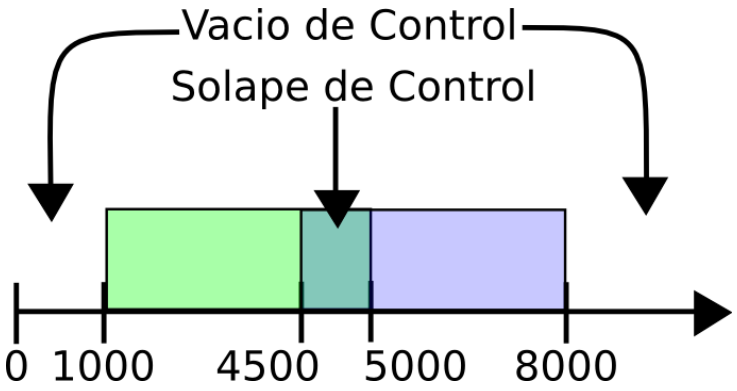


Figura 5.1: Regiones de control de los esquemas `hijo1` e `hijo2`

Para mostrar cómo lo hemos desarrollado, veremos cada uno de los pasos seguidos.

Inicialización de los hijos

Todos los esquemas son por defecto hijos del esquema ficticio `GUIHUMAN`, es decir, inicialmente no tienen constancia de quién será su padre. En este ejemplo, es el usuario humano quien debe proceder a activar al esquema `padre` mediante `schema_startup()`. Es en esta función en la que se informa al censo de esquemas de que `hijo1` e `hijo2` son hijos de `padre`. Además, se debe proporcionar la función de arbitraje que resolverá conflictos a los hijos, y se debe también proporcionar el identificador del grupo al que pertenece cada hijo. Finalmente se inicia la ejecución de los hijos.

```
G_MODULE_EXPORT void schema_startup (void) {
   ...
   function hijo1_startup, hijo2_startup;

   if (!loaded) {
      ...
   }
   else {
      schemaSetFather("hijo1", schema_id);
      schemaSetFather("hijo2", schema_id);

      schemaSetArbitration("hijo1", schema_arbitration);
      schemaSetArbitration("hijo2", schema_arbitration);

      schemaSetGroup("hijo1", 0);
      schemaSetGroup("hijo2", 0);
```

```
      hijo1_startup = schemaGetStartup("hijo1");
      hijo1_startup();
      schemaSetState("hijo1", checking);
      hijo2_startup = schemaGetStartup("hijo2");
      hijo2_startup();
      schemaSetState("hijo2", checking);
      ...
   }
}
```

Función principal de los hijos

La "maquina de estados" se ha materializado, al igual que en `jde.c`, dentro de la función principal de los esquemas hijo. Esta función es invocada sucesivas veces por el *timeout* iniciado en `schema_startup()`. Su cometido es comprobar a qué estado debe pasar, y si es finalmente *activo*, entonces procede con la ejecución de `schema_iteration()`.

```
gboolean schema_thread(gpointer data) {
   if (schemaGetState(schema_id) != slept) {
      if (precondiciones()) {
         schemaSetState(schema_id, ready);
         if (!schemaIsAnyWinner(schema_id))
            schemaSetState(schema_id, winner);
         else
            if (myarbitration(schema_id))
               schemaSetState(schema_id, winner);
      }
      else {
         schemaSetState(schema_id, checking);
         if (!schemaIsAnyReady(schema_id))
            if (myarbitration(schema_id))
               schemaSetState(schema_id, winner);
      }

      if (schemaGetState(schema_id) == winner)
         schema_iteration();
   }
   return TRUE;
}
```

Precondiciones y función de arbitraje

Los esquemas hijos deben tener una función que notifique si se cumplen las precondiciones, que en este ejemplo se trata de comprobar si el valor medido en `laser[90]` se encuentra en el rango esperado. Por ejemplo, la función que comprueba las precondiciones de hijo1 es la siguiente:

```
gboolean precondiciones() {
   return (laser[90] >= 4500 && laser[90] <= 8000);
}
```

La función de arbitraje de `padre`, que resuelve los casos de solape de control y de vacío de control, tiene la siguiente semántica: Si hay solape de control, ganará el control `hijo1`. Si hubiera vacío de control ninguno de los dos debe ejecutarse, y se procede a detener los motores:

```
gboolean schema_arbitration(gchar *id) {
   if (!schemaIsAnyReady("hijo1")) {
      v = 0.0;
      return FALSE;
   }
   else
      if (strcmp(id, "hijo1"))
         return TRUE;
      else
         return FALSE;
}
```

Resultado final

Hemos comprobado empíricamente el comportamiento de la infraestructura y los esquemas ante situaciones de solape de control y de vacío de control. Todos los componentes han reaccionado de la forma esperada.

5.2.3. Esquemas con interfaz gráfica

Para simplificar el desarrollo y el aprendizaje de la programación gráfica de esquemas, se ha hecho especial hincapié en el empleo del diseñador de interfaces Glade (Ver sección 3.1.1). Glade almacena toda la información relativa a la interfaz gráfica en un fichero XML, por lo que los esquemas que lo empleen tienen que poder acceder a dicho fichero. Por ese motivo, incorporamos la variable `schema_path` a la API de esquemas.

Además, como ya vimos en la sección 4.2.2, los esquemas que contengan interfaz gráfica deben necesariamente definir las funciones `schema_gui_startup()` y `schema_gui_suspend()`.

En el ejemplo propuesto crearemos el esquema `schema_gui`, que define una interfaz con una

etiqueta de texto y un botón. La función `schema_iteration()` modificará el valor de la etiqueta en cada iteración, y al pulsar el botón mostraremos por consola un aviso.

Inicialización y detención de la interfaz

En la primera ejecución de `schema_gui_startup()` se inicializará toda la interfaz gráfica. Cargaremos el fichero Glade que contiene la definición de la ventana, y asociaremos las señales a sus manejadores. En el resto de llamadas a `schema_gui_startup()` se visualizará la ventana principal.

```
G_MODULE_EXPORT gboolean schema_gui_startup () {
   GtkWidget *win, *button;

   if (gui_started) {
      g_print ("--- Inicializando gui del esquema %s---\n\n", schema_id);
      win = glade_xml_get_widget(schema_gui, "window1");
      gtk_widget_show(win);
   }
   else {
      gui_started = TRUE;

      schema_gui = glade_xml_new (g_strconcat(schema_path,
         "/", pathgui, NULL), NULL, NULL);
      if (!schema_gui)
         g_error("Error. No se ha inicializado correctamente
            la interfaz del esquema %s\n", schema_id);

      /* Atamos los callbacks */
      button = glade_xml_get_widget(schema_gui, "button");
      g_signal_connect (G_OBJECT (button), "clicked",
                     G_CALLBACK (on_button_clicked), NULL);
   }
   return TRUE;
}
```

Igualmente, la función encargada de detener la visualización se encargará de ocultar la ventana principal. De ese modo, todos los recursos quedan liberados gracias a que GLib no espera ningún evento sobre ellos.

```
G_MODULE_EXPORT gboolean schema_gui_suspend () {
   GtkWidget *win;
   g_print ("--- Suspendiendo gui del esquema %s ---\n\n", schema_id);
   win = glade_xml_get_widget(schema_gui, "window1");
   gtk_widget_hide(win);
```

```
   return TRUE;
}
```

Manejando la interfaz

En la función `schema_gui_startup()` asociamos el evento "clicked" a la función `on_button_clicked()`. En este ejemplo, esta función manejadora tan sólo enviará un aviso por consola, para dejar constancia de su correcto funcionamiento:

```
void on_button_clicked(GtkButton *button, gpointer user_data) {
   g_print("Boton pulsado\n");
}
```

Para comprobar cómo puede un esquema modificar en tiempo de ejecución sus *widgets* hemos programado una función `schema_iteration()` que modificar el valor de la etiqueta de texto de la interfaz gráfica.

```
gboolean schema_iteration(gpointer data) {
   static int i = 0;
   gchar label_text[20] = "";
   sprintf(label_text,"iteracion %d",i++);
   gtk_label_set_text(GTK_LABEL(glade_xml_get_widget(
      schema_gui, "label")), label_text);
   return TRUE;
}
```

Al ejecutar el esquema hemos comprobado cómo cada uno de los diferentes componentes programados se comporta según lo esperado. Por tanto, damos por resuelta la programación de la infraestructura que permite a los esquemas contener su propia interfaz gráfica.

5.3. Ejemplo real con esquemas

Finalmente, hemos desarrollado un ejemplo real que, a la vez que pone en práctica todas funcionalidades de la infraestructura, también realiza un algoritmo de navegación segura. El comportamiento deseado es el siguiente: estableceremos un destino, y el robot debe navegar hasta él esquivando los posibles obstáculos que pueda haber.

Se ha desarrollado una serie de esquemas, `avance`, `vff` y `stop`, que gobernados por un padre, trabajan en conjunto para controlar el robot y lograr que llegue al objetivo. Dependiendo de la distancia a los obstáculos se ejecutará un esquema hijo u otro. Cada uno de ellos tiene sus propias precondiciones, y hay casos en los que surgirá solape de control.

Si algún obstáculo entra en la zona de activación de `parada` entonces será candidato a ser ejecutado. Del mismo modo, si algún obstáculo entra en el área de activación de `vff`, este

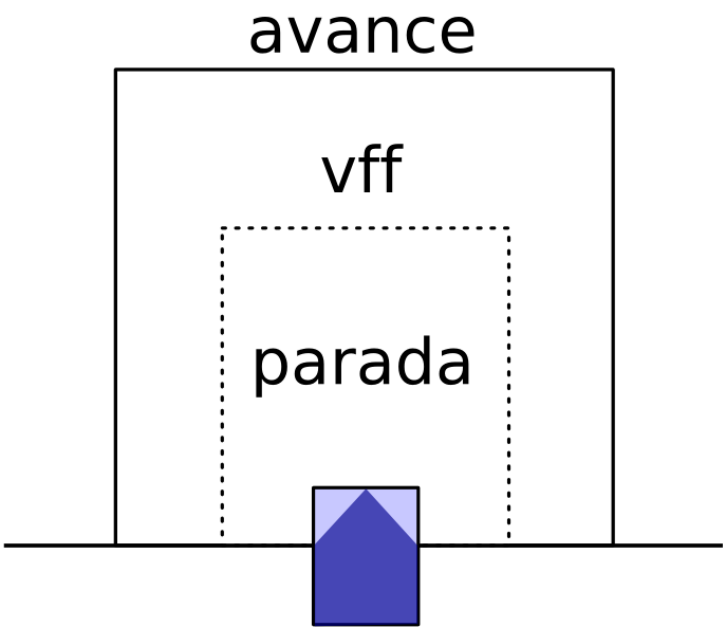


Figura 5.2: Regiones de control de los esquemas de navegación

esquema podrá obtener el control. Vemos en la figura 5.2 que el conjunto de estímulos que activa el esquema `parada` está incluido dentro de `vff`. En este caso la función de arbitraje determinará que debe ser `parada` quién gane el control. Finalmente, si no hay ningún obstáculo en las áreas de activación de `vff` y de `parada`, entonces se ejecutará `avance`.

Parada

El esquema de parada tiene como precondiciones la detección de obstáculos en su área de activación. Este área debe ser muy reducida, ya que el robot sólo debe detener su ejecución en aquellos casos en que los obstáculos sean tan cercanos que puedan suponer un choque. La acción a llevar a cabo es la detención de la base motora del robot.

Avance

Este esquema se encarga de navegar a ritmo ligero. Sus precondiciones son no detectar obstáculos en las áreas de activación de `vff`. La acción a realizar es por tanto, modificar la velocidad frontal `v` y la velocidad angular `w` para dirigirse al punto de destino. Los valores asignados se obtienen a partir del ángulo del vector que tiene como origen el robot y como destino las coordenadas del objetivo.

VFF

El esquema de navegación `vff` empleará una técnica conocida como *campos de fuerza virtuales* (Virtual Force Fields [Borenstein y Koren, 1989]). Este mecanismo genera a partir de los obstaculos un vector en las coordenadas del robot que representa lo que llamaremos *fuerza de repulsión*. Cuanto menor es la distancia medida al obstáculo, mayor importancia se le da, y mayor fuerza de repulsión ejerce. El destino, por otra parte, generará un vector que llamaremos *fuerza de atracción*. El sumatorio de ambas fuerzas será la *fuerza resultante* (Ver figura 5.3).

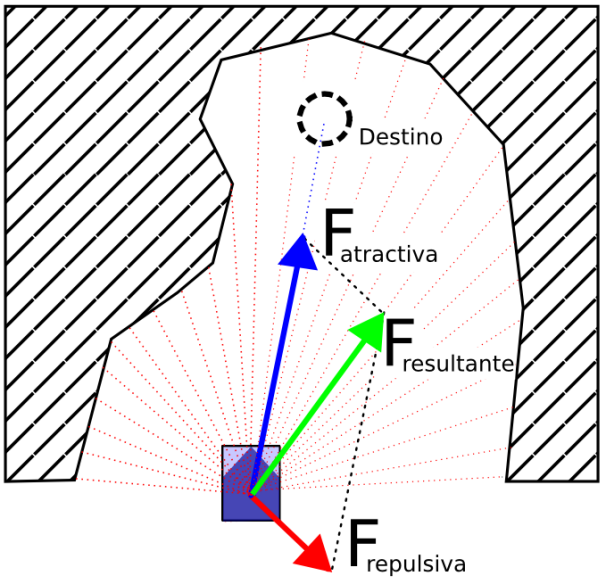


Figura 5.3: Fuerzas vectoriales en navegación VFF

Dependiendo del valor de la fuerza resultante se realizará una acción u otra. Atendiendo al ángulo de la fuerza resultante respecto del robot hemos considerado los siguientes casos de acción:

1. $Angulo \leq \pm 10^{o} \Rightarrow$ Velocidad frontal alta. Velocidad angular nula.
2. $\pm 10^{o} < Angulo \leq \pm 40^{o} \Rightarrow$ Velocidad frontal media. Velocidad angular media.
3. $Angulo > \pm 40^{o} \Rightarrow$ Velocidad frontal nula. Velocidad angular alta.

Implementación

En el desarrollo de este conjunto de esquemas hemos puesto en práctica la mayor parte de funcionalidades ofrecidas por la plataforma. Por un lado, el esquema principal proporciona al usuario una interfaz gráfica para poder depurar el esquema y para poder seleccionar el punto de destino. Por otra parte hemos empleado una jerarquía de un grupo de tres hermanos: `vff`, `parada` y `avance`. Finalmente, también hemos usado la comunicación entre el esquema padre y los esquemas hijo `vff` y `avance`. Estos hijos admiten modulación a su comportamiento, estableciendo el padre las coordenadas del punto de destino, variables en las que ellos leen.

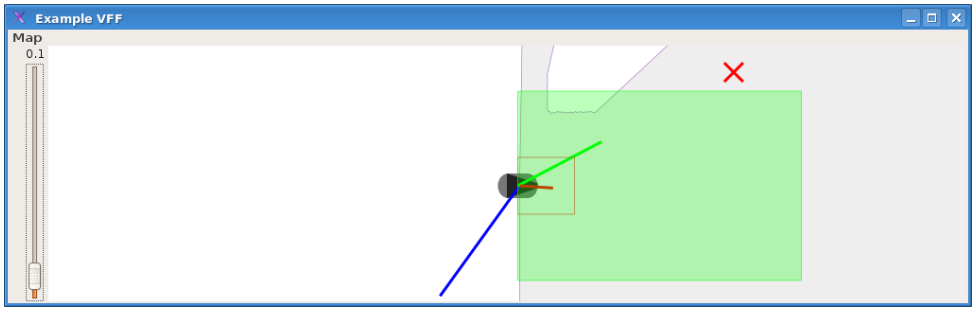


Figura 5.4: Visualización del esquema de navegación

Interfaz de usuario

Para poder visualizar el entorno que detectan los sensores del robot, y para poder establecer las coordenadas del punto de destino, hemos creado una sencilla interfaz gráfica gobernada por el esquema padre. Una vez seleccionadas las coordenadas del objetivo, se muestran también las fuerzas vectoriales correspondientes a la navegación del esquema `vff`, y las superficies que delimitan las áreas de acción de cada esquema (Ver figura 5.4).

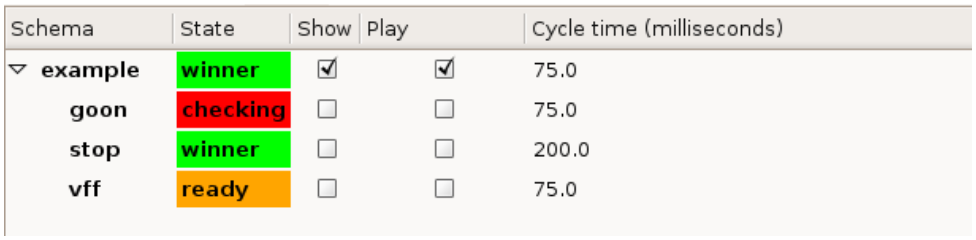


Figura 5.5: Jerarquía de esquemas de navegación

Capítulo 6

Conclusiones y trabajos futuros

Una vez descrita la solución aplicada para resolver las motivaciones fijadas, y vistos algunos de los resultados experimentales más relevantes, finalizaremos esta memoria evaluando los resultados obtenidos y concretando las futuras líneas de trabajo que surgirán a raíz de `jdeneo.c`. En esta sección resumiremos la conclusiones sacadas del desarrollo de este proyecto. Enumeraremos algunos de los subobjetivos planteados en el capítulo 2 y comentaremos nuestra impresión personal sobre el resultado obtenido.

6.1. Conclusiones

El objetivo de este proyecto ha sido crear `jdeneo.c`, una nueva plataforma de desarrollo de aplicaciones robóticas inspirado en la arquitectura cognitiva JDE. Hemos partido de la versión 3.4 de `jde.c`, la cual, pese a ser una infraestructura muy madura, presentaba deficiencias en algunos aspectos. `jdeneo.c` ha tratado de ser un nuevo desarrollo que solventara la mayoría de esas deficiencias. Las más destacables han sido el diseño empleando la biblioteca de propósito general `GLib`, el rediseño de esquemas ofreciendo una nueva API y permitiendo cargas en tiempo de ejecución, implementar la interfaz gráfica haciendo uso de la biblioteca `GTK+`, y ofrecer visualización 3D de la escena que rodea al robot.

Uno de los hitos correspondía a diseñar la infraestructura empleando la biblioteca de propósito general GLib (Ver sección 3.2). A lo largo de todo el desarrollo se ha empleado dicha herramienta, haciendo uso de la capa de abstracción que la plataforma GLib nos ofrece. En un principio supuso conocer en profundidad todos los mecanismos que GLib nos aporta, adaptándonos a la programación empleando su metodología, y evitando, de ese modo, el uso de tipos y funciones estándar en C. Uno de los mayores logros derivados del uso de `GLib` ha sido crear carga dinámica de esquemas, empleando la herramienta `GModule`. Gracias a ello, hemos logrado un comportamiento de *plugins* que evita la recompilación de toda la infraestructura con cada inserción o modificación de esquemas. Otro gran avance ha sido el uso de los *timeouts*, que han permitido delegar los mecanismos de alarmas sobre herramientas muy sólidas. GLib ha aportado a la infraestructura un núcleo robusto, fiable y escalable.

Probablemente el subobjetivo que ha resultado más costoso en tiempo haya sido el diseño y programación de la interfaz gráfica. Al emplear las bibliotecas gráficas GTK+ (Ver sección 3.1), ha sido necesario aprender previamente GLib,. Como vimos en la sección 3.1.1, el uso del

asistente de diseño de interfaces Glade alivió gran parte del diseño, ofreciendo un método visual óptimo para el aprendizaje. Sin embargo, hay muchos elementos que Glade no contiene, por lo que ha sido necesaria una programación híbrida de la interfaz: Por un lado tenemos el diseño realizado en Glade (Como por ejemplo la interfaz principal) y por otro el diseño creado desde código (Como la ventana de visualización 3D o el árbol desde el que se muestra y gobierna la jerarquía de esquemas). Si bien esta solución ha sido muy favorable para el aprendizaje de GTK+, no ha sido elegante, y además dificulta la comprensión de la estructuración del código, confundiendo en el aprendizaje de posibles nuevos desarrolladores de `jdeneo.c`.

Podemos valorar como óptimo el método seguido para el aprendizaje del diseño de interfaces en GTK+. Hemos trabajado con un gran número de *widgets*, hemos desarrollado interfaces desde dos vías distintas (cada una con sus pros y sus contras) y hemos aprendido a extender los controles de GTK+ añadiendo bibliotecas auxiliares.

La programación de la infraestructura ha estado, en general, más relacionada con ingeniería del software que con generación de comportamientos sobre robots. Debido a eso, uno de los objetivos más interesantes ha sido la implementación de la arquitectura de esquemas propuesta en la infraestructura teórica de JDE, ya que hemos tenido la oportunidad de comprender y poner en práctica todos los mecanismos necesarios, y ver cómo se coordinan los esquemas para lograr exitosamente los objetivos.

Otro de los hitos fue, como vimos en el apartado 4.3, la programación de la visualización 3D. Para ello fue necesario un periodo de aprendizaje de OpenGL, que nos permitiera conocer los mecanismos que ofrece la plataforma en el desarrollo de escenas tridimensionales. Como consecuencia del bajo nivel que ofrecen las primitivas de OpenGL, fue necesario aprender a manejar el diseñador 3D Blender, para posteriormente exportar los objetos a código fuente. Finalmente se procedió a la integración de todos los componentes implicados en la visualización. El resultado final ofrece al usuario una escena muy vistosa y rica en detalles, que permite de forma nativa generar comportamientos en mundos en 3D.

Finalmente, con el desarrollo del ejemplo real y de las pruebas de mecanismos, hemos podido comprobar si el comportamiento del conjunto de la arquitectura es coherente. Los mecanismos de comunicación entre esquemas han funcionado satisfactoriamente, la implementación de la jerarquía se ha comportado en todo momento según los requisitos fijados, y también hemos podido verificar que la distribución de la visualización de interfaces gráficas se comporta según lo esperado. Además, con el ejemplo real, hemos podido implementar un comportamiento que requiere del correcto funcionamiento de todos los componentes de la infraestructura.

6.2. Trabajos futuros

El desarrollo de `jdeneo.c` está abierto a nuevas mejoras y tiene algunas cuestiones sensibles a ser mejoradas o ampliadas. Algunos de ellos ya está previsto realizarlos, mientras que

otros son posibles mejoras para una futura continuación con el desarrollo de esta infraestructura.

`jde.c`, la aplicación que sirvió como embrión para `jdeneo.c`, es un proyecto vivo y en continuo desarrollo. Desde el inicio de este proyecto, `jde.c` ha seguido evolucionando, especialmente en la organización de la infraestructura y en la creación de numerosos *drivers*, logrando un entorno de desarrollo maduro y muy robusto. Todos los conocimientos aprendidos y puestos en práctica sobre GTK+ serán agregados a la siguiente versión oficial de `jde.c`, adaptándola, por tanto, a bibliotecas de última generación.

A lo largo del desarrollo hemos tenido presente que una de las ventajas que nos ofrece la biblioteca de propósito general GLib es la posibilidad de crear aplicaciones multiplataforma, abstrayéndonos de ese modo de la máquina y del sistema operativo sobre el que está funcionando la aplicación. `jdeneo.c` actualmente no cumple todos los requisitos para poder ser una aplicación multiplataforma, ya que depende fuertemente de bibliotecas exclusivas de GNU/Linux. Sin embargo, al haber empleado GLib como tecnología de bajo nivel que subyace a gran parte del código de `jdeneo.c`, la tarea de adaptación para crear una aplicación multiplataforma ya está comenzada.

Una posible mejora sería modificar las estructuras en las que almacenamos toda la información relevante a los esquemas, el *censo de esquemas* (Ver sección 4.2.3) y a las variables compartidas, el *censo de variables*. Hemos empleado listas enlazadas, que si bien permiten tener un número indefinido de *items*, la tarea de localizar un elemento en concreto tiene un coste computacional alto, ya que debe recorrer uno a uno todos los *items* hasta encontrar el deseado (Complejidad lineal: O(n)). En ese contexto la solución más acertada habría sido emplear tablas *hash*, con lo que obtenemos una complejidad media de búsqueda constante, O(1), lo que mejora sustancialmente el coste computacional respecto a la listas enlazadas. No obstante, esta modificación no supondría un notable descenso en el uso del procesador debido a que los accesos a las estructuras dinámicas se han acotado, y se realizan en contadas ocasiones a lo largo de la ejecución de la infraestructura.

Otra posible mejora consistiría en implementar facilidades para el diseño de esquemas. Hemos podido comprobar cómo gran parte de la funcionalidad que incorporan los esquemas se repite en muchos de ellos. Por ejemplo, la máquina de estados de los esquemas de actuación es prácticamente igual en todos ellos. Una posible solución habría pasado por incorporar todas las tareas "estándar" dentro del núcleo de la plataforma, simplificando notablemente la programación de aplicaciones con comportamientos.

Bibliografía

[Borenstein y Koren, 1989] J. Borenstein y Y. Koren. Real-time obstacle avoidance for fast movile robots. *IEEE Journal of Robotics and Automation*, 1989.

[Can, 2007] Gnome canvas library reference manual. `http://developer.gnome.org/doc/API/2.0/libgnomecanvas/index.html`, 2007.

[Cañas *et al.*, 2007] José María Cañas, David Lobato, y Pablo Barrera. jde+: an open-source schema-based framework for robotic applications. *Universidad Rey Juan Carlos de Madrid*, 2007.

[Gerkey *et al.*, 2000] Brian Gerkey, Kasper Støy, y Richard T. Vaughan. Player robot server. *Institute of Robotics and Intelligent Systems, School of Engineering, University of Southern California*, 2000.

[GTK, 2007a] Glib reference manual. `http://developer.gnome.org/doc/API/2.0/glib/`, 2007.

[gtk, 2007b] Gtkglext reference manual. `http://gtkglext.sourceforge.net/reference/gtkglext/`, 2007.

[Hemstridge, 2002] James Hemstridge. Libglade reference manual. `http://developer.gnome.org/doc/API/2.0/libglade/index.html`, 2002.

[Isado, 2005] José Raúl Isado. Navegación global utilizando método del gradiente. *Proyecto Fin de Carrera, URJC*, 2005.

[Lobato, 2005] David Lobato. Arquitectura jde+ para el control de robots. *Proyecto Fin de Carrera, URJC*, 2005.

[Maya, 2006] Ricardo Palacios Maya. Representación rica de la escena 3d alrededor de un robot móvil. *Proyecto Fin de Carrera, URJC*, 2006.

[Moya *et al.*, 2002] Rodrigo Moya, Álvaro del Castillo, Roberto Majadas, Gaspar Oriol, Gregorio Robles, Yolanda Moreno, Ayose Cazorla, et al. Programación en el entorno gnome. *The GNOME Hispano*, 2002.

[Neider *et al.*, 1993] Jackie Neider, Tom Davis, y Mason Woo. *OpenGL Programming Guide*. Addison-Wesley Publishing Company, 1993.

[Pennington, 1999] Havoc Pennington. *GTK+ / Gnome Application Development*. New Riders Publishing, 1999.

[Plaza *et al.*, 2007] José María Cañas Plaza, Vicente Matellán, y Rodrigo Montúfar. Programación de robots móviles. *Universidad Rey Juan Carlos de Madrid e Instituto Nacional de Astrofísica, Óptica y Electrónica, México*, 2007.

[Plaza, 2003] José María Cañas Plaza. *Jerarquía Dinámica de Esquemas para la generación de comportamiento autónomo*. PhD thesis, Universidad Politécnica de Madrid, 2003.

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

[Vaughan, 2000] Richard T. Vaughan. Stage: A multiple robot simulator. *Institute of Robotics and Intelligent Systems, School of Engineering, University of Southern California*, 2000.

[Wik, 2006] Artículo de introducción a los robots. `http://en.wikipedia.org/wiki/Robot`, 2006.