Planificación de movimientos en robótica: curso práctico para implementar un PRM

Alexander Pérez* y Jan Rosell
Institut d'Organització i Control de Sistemes Industrials
Universitat Politècnica de Catalunya
alexander.perez@upc.edu, jan.rosell@upc.edu

Resumen

Los trabajos de investigación en el área de la planificación de movimientos para robots necesitan disponer de entornos de simulación para vaildar las contribuciones teóricas. Sin embargo los programas de postgrado y doctorado de robótica suelen abordar el tema sólo desde el punto de vista teórico. Por ello los estudiantes que quieren enfocar sus investigaciones en esta linea, deben asumir el gran desafío de desarrollar su propios sistemas de simulación. Para mitigar esto, se presentan en este artículo doce prácticas de programación en las que paulatinamente se desarrolla un entorno de trabajo para la planificación de movimientos y la simulación de robots basado en herramientas multiplataforma que siguen la filosofía de software libre. Este entrenamiento cubre los tópicos básicos de la representación gráfica, la detección de colisiones y el manejo de grafos y sus algoritmos.

Palabras clave: Planificación, Movimientos, Robótica, Simulación .

1. INTRODUCCIÓN

Los programas de postgrado en robótica, incluyendo los de doctorado, normalmente incluyen la planificación de movimientos entre sus cursos troncales. Estos cursos suelen tratar temas básicos de la simulación de robots tales como el modelado de objetos, la detección de colisiones y otros problemas de geometría computacional, la representación y manipulación de grafos, la representación gráfica tridimensional y las particularidades de las interfaces de usuario. La enseñanza de estos temas es clave para cerrar la brecha existente entre la teoría y la práctica. Adquirir destreza en estos temas, permitirá a los estudiantes enfocarse en la prueba y validación de sus contribuciones teóricas.

En Internet se encuentran varios planificadores de software libre disponibles, tales como *"An Object-Oriented Programming System for Motion Planning"* (OOPSMP, www.kavrakilab.org/software), el *"Motion Planning Kit"* (http://ai.stanford.edu/mitul/mpk) o el *"Motion Strategy Library* (http://msl.cs.uiuc.edu/msl/). A pesar de que poseen características interesantes, estos no cubren las espectativas que se tienen desde el punto de vista didáctico. No son fáciles de usar y es difícil adentrarse en su estructura de manera que se pueda adaptar y extender a las necesidades particulares. Es por ello que incluir tópicos de la simulación dentro de los cursos de planificación de movimientos es interesante y brinda a los alumnos la oportunidad de enfrentarse a los diferentes problemas relacionados, así como adquirir una visión global del campo de trabajo.

El primer objetivo de este artículo es proponer un marco de programación que cubra la mayor cantidad de aspectos de la simulación de robots, necesarios para el desarrollo de software de planificación de movimientos. Teniendo presente la filosofía de software libre y la característica de ser multi-plataforma, la sección 2 presenta el conjunto de herramientas de programación que componen este marco. El segundo objetivo es proponer un método para adquirir de forma rápida y sencilla las habilidades necesarias para trabajar con ellas. La sección 3 presenta un conjunto de doce prácticas de programación que de forma incremental y a manera de bloques constructivos, van cubriendo los diferentes aspectos del desarrollo de un planificador de movimientos. En la primera práctica se comienza con un programa sencillo que modela entidades geométricas básicas y se termina en la última práctica con una versión básica de un planificador de tipo *"Probabilistic Roadmap"* para robots *"Free-Flying"*. Finalmente la sección 4 concluye el trabajo.

2. HERRAMIENTAS

2.1. Herramientas de desarrollo de software

El desarrollo de una aplicación de planificación de movimientos en robótica requiere del correcto uso de diferentes herramientas de software [11]. Estas incluyen la necesidad del control de versiones para

* Escuela Colombiana de Ingeniería "Julio Garavito", Bogotá D.C., Colombia

propósitos de seguimiento y corrección de errores y una forma fácil de generar una buena documentación. Además, para una buena difusión, es interesante que el desarrollo sea versátil y pueda ser ejecutado en diferentes plataformas y que requiera sólo de librerías de software libre.

Paradigmas de programación y lenguajes: Uno de los paradigmas de programación más publicados es el Orientado a Objetos (POO, [3]), que permite un confortable trabajo con modelos (clases) de todos los actores involucrados en el problema, como el espacio de configuraciones, el espacio de trabajo, los obstáculos y los robots en el caso de la planificación de movimientos. La POO permite crear código modular y reutilizable. Un paso más allá hay el paradigma de la programación genérica que extiende y brinda mayor flexibilidad a a la POO. Al desarrollar software siguiendo los lineamientos del la POO, puede ser muy útil expresar de forma gráfica su diseño usando el Lenguaje de Modelado Unificado [2]. Herramientas gráficas como el StarUML (www.staruml.com) permiten hacer uso del UML para realizar un diseño inicial a nivel de clases que luego puede ser refinado o modificado.

Una elección extendida es el uso del C++ como lenguaje de programación, que cumple adecuadamente con los lineamientos de la POO y ademñas existe un compilador disponible para todas las plataformas. En particular el *cl* para *Windows* (www.microsoft.com/express/download) y el *gcc* para *Linux* (http://gcc.gnu.org).

Ambos compiladores son de línea de comandos, y para facilitar su uso en los procesos de construcción de las aplicaciones normalmente traen una utilidad, que para el caso de *Windows* es *nmake* y para *Linux* es *make*. Con la intención de liberar al programador de las características dependientes de la plataforma se usa un sistema multiplataforma y de software libre denominado *CMake* (www.cmake.org, [10]) que se encarga de la configuración del proyecto a partir de un archivo de texto genérico. Las instrucciones de compilado y construcción se encuentran codificadas en un lenguaje de tipo *"script"* dentro del archivo denominado CMakeList.txt.

Sistema de control de versiones: Subversion SVN (https://sub-version.tigris.org, [4]) es un sistema que ofrece un potente ambiente para almacenar y recuperar todas las sucesivas versiones de un archivo dentro de un proyecto a lo largo de todo su desarrollo. Es muy útil para realizar el seguimiento y corrección de errores y esencial si el desarrollo se realiza entre varios programadores. SVN ofrece también un conjunto de herramientas administrativas tanto para las operaciones de cliente como las de servidor. Una buena elección para el lado del servidor es OpenSVN (https://opensvn.csie.org), una iniciativa de software libre para proyectos de igual naturaleza. Para el lado del cliente, TortoiseSVN (http://tortoisesvn.net) es una aplicación que se integra dentro del explorador de *Windows* ofreciendo gran versatilidad. Para entornos *Linux*, kdesvn (http://kdesvn.alwinsworld.de) es una aplicación totalmente integrada al escritorio KDE (herramientas similares se pueden encontrar para otros escritorios).

Estas herramientas pueden ser complementadas con otras que permitan comparar dos versiones del mismo archivo permitiendo revisar y seguir los cambios (e.g. winmerge en www.winmerge.org o KDiff3 en http://kdiff3.sourceforge.net).

Sistema de documentación: Doxygen es un sistema de documentación multi-plataforma y multilenguaje, entre ellos C++ y *Java*. Es realmente fácil de utilizar dado que el programador sólo necesita agregar comentarios con una marca particular que es reconocida y permite generar automáticamente documentos tanto en Latex como en html.

2.2. Representación Gráfica e Interfaz de Usuario

Toda aplicación de software que interactúe con el usuario necesita una interfaz a través de la cual intercambie información con él. Una buena librería multi-plataforma con la que se construyen interfaces es *Qt* (http://trolltech.com/products/qt), ya que es utilizada en una gran variedad de aplicaciones tanto comerciales como de software libre y cuenta con la experiencia de mas de 10 años de constante desarrollo. Además cuenta con una sencilla utilidad (*QtDesigner*) que permite diseñar de forma gráfica la interfaz a través de *"drag and drop"* que facilita mucho este proceso creativo.

Para la representación gráfica de los modelos 3D, una buena alternativa es *Coin3D* (www.coin3d.org). Esta librería gráfica basada en las fuentes liberadas por Silicon Graphics de *OpenInventor*, es actualmente la implementación adoptada por todas las distribuciones *Debian Linux*. *Coin3D* ofrece una interfaz similar a un software de CAD. Con él se pueden manejar tanto modelos básicos como esferas, conos o cubos, o formas más generales modeladas a través de mallas triangulares. *Coin3D* puede trabajar conjuntamente con las librerías de creación de ventanas, que pueden ser *Qt*, *Motif/Xt*, *Windows* o *MacOS*, para lo cual es indispensable contar con las librerías *SoQt* o *Quarter*, *SoXt*, *SoWin* y *Sc21* respectivamente.

Cualquier modelo 3D puede ser descrito en un archivo de texto a través de los lenguajes *Inventor*

o *VRML* (http://www.w3.org/ MarkUp/VRML). Estos describen la escena gráfica con objetos de alto nivel como cubos, cilindros o esferas o cualquier grupo de estos, y/o con mallas triangulares. Adicionalmente cuenta con nodos de propiedades que pueden afectar la forma en la que éstos son dibujados. *Coin3D* puede cargar en la escena modelos provenientes de archivos tanto de *Inventor* como de *VRML*.

Una herramienta muy útil que permite crear rápidamente estas escenas 3D, es *Coindesigner* (http://coindesigner.sourceforge.net), que lo consigue usando *"drag and drop"* y visualizando los cambios en el acto.

Para la descripción completa de una escena robótica, el *XML* puede resultar de gran utilidad sobre todo cuando se usa bajo el esquema *SAX* (http://sax.sourceforge.net). El archivo *XML* puede contener información tanto del robot como de los obstáculos (ubicación del archivo *VRML* o *Inventor*, nombres, escalas, posiciones y orientaciones), y cualquier otro aspecto relacionado con la definición del problema de planificación como las configuraciones inicial y final. La librería *Qt* tiene implementado un lector de *SAX* que hace muy sencillo el proceso de lectura de estos archivos *XML*, además de contar con características interesantes para la manipulación de cadenas y lectura de archivos.

2.3. Herramientas de Geometría Computacional

Actualmente, los métodos basados en muestreo son los más comúnmente utilizados en la resolución de problemas de planificación de movimientos. Estos métodos se basan en la generación de muestras libres de colisión en el espacio de configuraciones (*Cspace*) y que en conjunto con las interconexiones, tambien libres, forman los *"Roadmaps"* (*PRM* [6]) o los árboles (*RRT* [7]). Varios de los tópicos necesarios para el desarrollo de un planificador basado en el muestreo son del campo de la geometría computacional: a) la ubicación del objeto; b) la generación de muestras; c) la determinación del estado de colisión o no de las muestras; d) la búsqueda de vecinos; e) la representación en grafos y sus algoritmos de búsqueda.

La determinación de la posición y orientación de los objetos en el espacio pueden acarrear varios problemas si no se realiza adecuadamente, e.g. debido a una ineficiente representación se pueden acarrear errores numéricos. La librería *mt* [13] brinda un gran soporte en este sentido al proporcionar clases que definen estas representaciones.

Las muestras pueden ser generadas usando tanto métodos aleatorios como determínisticos. La muestras aleatorias son generadas, asignándoles valores aleatorios a las coordenadas de la configuración, que normalmente son obtenidos a través de generadores de números seudo-aleatorios con períodos muy grandes y estadísticamente aceptados, como la tratada en [12] que ofrece mejores resultados que la clásica función *rand* de C. Las secuencias determinísticas ofrecen muestras con un mejor cubrimiento, incremental y uniforme, del *Cspace*. La secuencia Halton es una de las mejores secuencias de baja discrepancia (http://people.scs.fsu.edu/∼burkardt, [5]).

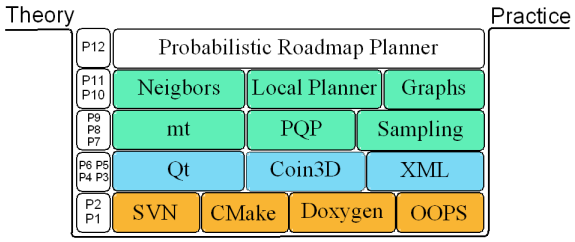

Figura 1: Conceptos cubiertos por el grupo de prácticas.

Los procedimientos de detección de colisiones prueban si el objeto de la escena se encuentra en colisión o no (i.e. si el robot en una determinada configuración choca o no contra un obstáculo). Se pueden encontrar varias librerías de detección de colisiopnes en Internet. Una buena alternativa, dada su sencillez, es *PQP "the Proximity Query Package"* (www.cs.unc.edu/geom/SSV, [8]), que viene siendo usada actualmente por varios planificadores. Esta basada en rápidos y confiables algoritmos que encuentran colisiones o distancias entre objetos descritos por mallas triangulares, que a su vez pueden ser obtenidos de los modelos *Inventor/VRML* a través de procedimientos proporcionado por *Coin3D*.

Con el fin de conectar las muestras obtenidas y capturar la conectividad de la parte libre del *Cspace*, es necesario encontrar para cada muestra, cuál es su vecino más cercano. Está operación puede ser eficientemente resuelta usando la librería *ANN* (www.cs.umd.edu/∼mount/ANN, [1]). La *ANN* es una librería escrita en C++, que aporta las estructuras de datos y los algoritmos para buscar tanto de forma exacta como aproximada, los vecinos de una muestra en un espacio que puede ser de muchas dimensiones. Una adaptación de esta librería a diferentes topologías de *Cspace*, con su correspondiente métrica está disponible en msl.cs.uiuc.edu/∼yershova, [15].

Para manejar estructuras de datos tipo grafo, una buena alternativa por su eficiencia es la librería

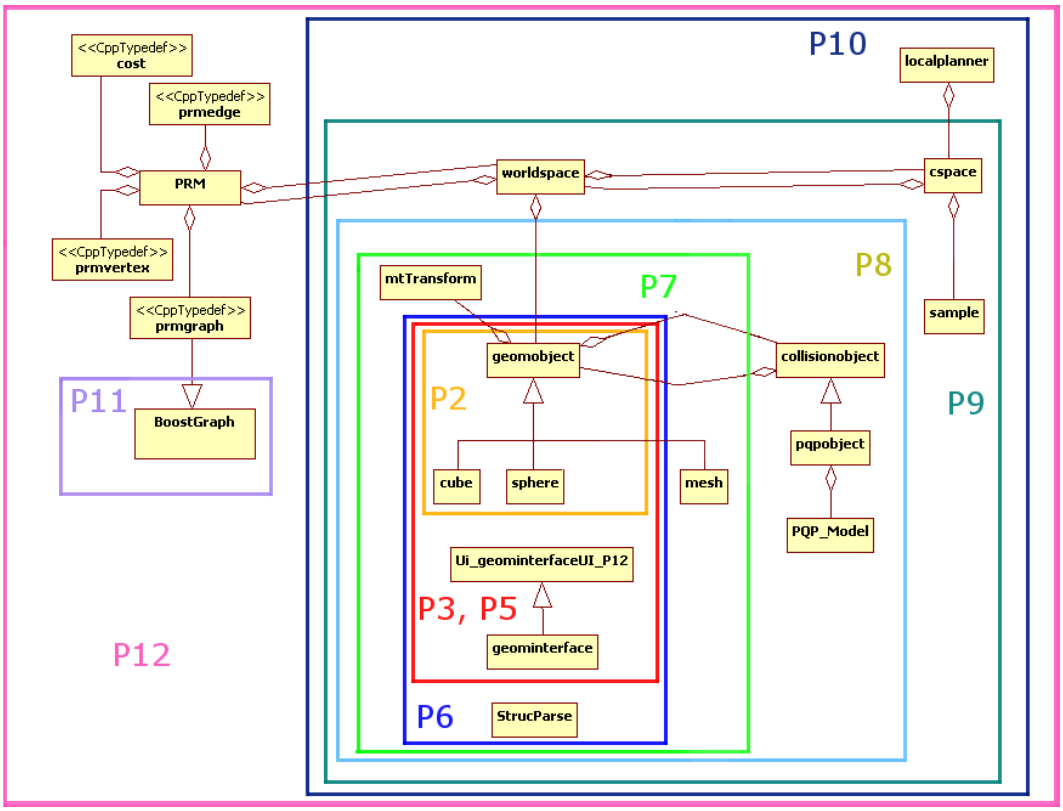

Figura 2: Diagrama UML de clases de todo el proyecto.

Boost Graph (www.boost.org/libs/graph, [9]), que está implementada con el paradigma de la programación genérica. Proporciona una interfaz generalizada estándar con *"Templates"* de los algoritmos de búsqueda como los de profundidad, anchura, costo uniforme o A*, y es fácilmente adaptable a cualquier aplicación particular.

3. PRÁCTICAS DE PROGRAMACIÓN

En esta sección se propone un conjunto de doce prácticas de programación para ser usadas como contraparte a la teoría dentro de un curso de planificación de movimientos en robótica. Se asume que: a) el curso dura un semestre y que introduce entre otros los temas de espacio de configuraciones, técnicas basadas en muestreo y el concepto del espacio de configuraciones; b) los estudiantes tienen buenas bases de programación en C++.

Cómo se muestra en la Fig. 1, las prácticas de programación cubren de forma incremental todos los aspectos básicos de la construcción de un planificador probabilístico de tipo PRM. La Fig. 2 muestra el diagrama UML de las clases que componen el PRM agrupadas de acuerdo a la práctica en la que fueron introducidas. Las siguientes subsecciones presentan cada una de estas prácticas resaltando de cada una el objetivo, el material proporcionado y la tarea propuesta para los alumnos, basada en modificaciones menores, que les permita una rápida y sencilla apropiación de los conocimientos.

Antes de comenzar con las prácticas, los estudiantes deberán instalar los siguientes programas en sus ordenadores: CMake, TortoiseSVN (si se usa *Windows*), *Doxygen*, *Coin3D*, *Qt4*, *SoQt*, *mt*, *PQP*, la librería *"Boost Graph"* y el *CoinDesigner2*. Una guía de descarga e instalación, así como las doce prácticas, se pueden encontrar de manera gratuita en http://iocnet.upc.edu/usuaris/JanRosell/EducationalTools.html.

3.1. Práctica P1: Aprendiendo a usar SVN y CMake

Objetivo: En esta primera práctica, se propone el acercamiento de los estudiantes a las herramientas de control de versiones y de generación de los proyectos (de VS o el MakeFile) usando SVN y CMake.

Material Disponible: Algunos ejemplos simples.

Tarea: Los estudiantes harán el *"checkout"* correspondiente y ejecutarán el CMake para crear los

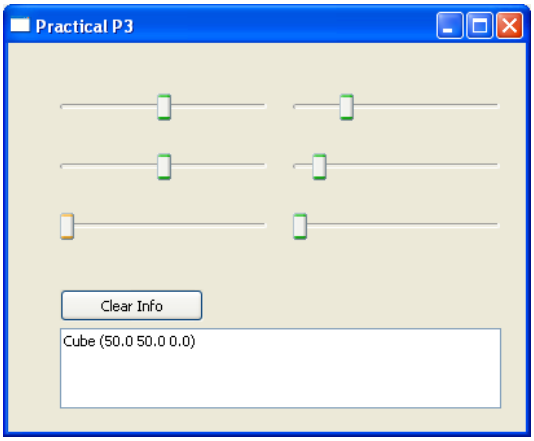

Figura 3: GUI de la práctica P3: Interfaz sencilla en Qt con controles que cambian el punto de referencia de los dos objetos de la escena.

proyectos, tanto en *Windows* como en *Linux*, de todos los ejemplos mantenidos en el repositorio como P1. Luego se procederá a compilarlos, construirlos y ejecutarlos.

3.2. Práctica P2: Documentando con Doxygen.

Objetivo: Aprender cómo se debe documentar el código y generan los documentos a través del sistema *Doxygen*.

Material Disponible: Un proyecto sencillo que muestra una clase abstracta denominada geomobject de la que se derivan las clases: cube y sphere. Estas clases están organizados en carpetas separadas y compiladas como la librería libGeometry. Aquí se muestran los conceptos de polimorfismo y programación genérica usando *"Templates"*, característico de la POO.

Tarea: Extender la jerarquía con dos nuevas clases denominadas cone y cylinder, y documentarlas adecuadamente usando *Doxygen*.

3.3. Práctica P3: Una sencilla interfaz usando Qt.

Objetivo: Usar *QtDesigner* para modificar una interfaz y aprender cómo son manejados sus eventos a través del mecanismo de *"Signals/Slots"*.

Material Disponible: El proyecto P2 ampliado con una interfaz de *Qt* compuesta por seis barras de deslizamiento y un cuadro de texto. Las primeras sirven para cambiar el punto de referencia de los dos objetos instanciados de las clases cube y sphere y el segundo los muestra (Fig. 3).

Tarea: Completar la interfaz de *Qt* adicionando barras de deslizamiento que permitan cambiar la

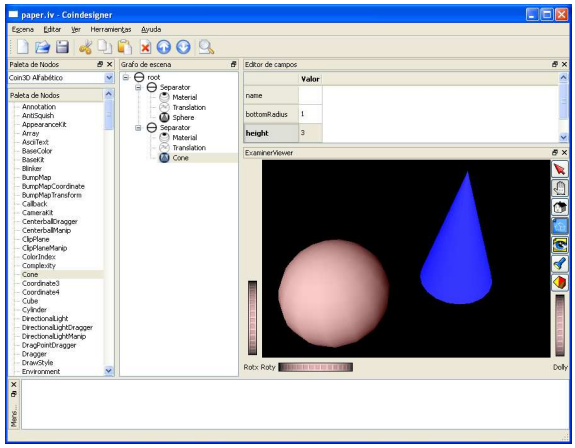

Figura 4: Escena Inventor creada y visualizada con CoinDesigner2.

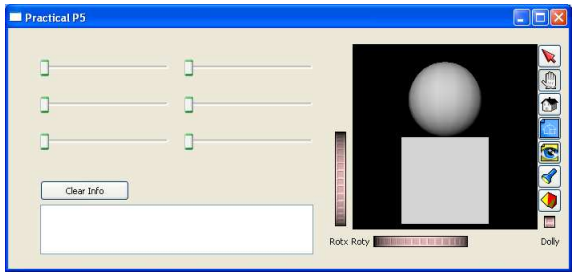

Figura 5: Interfaz de P5: Visualización de la escena usando *Coin3D*.

información de dos nuevos objetos instanciados de las clases cone y cylinder.

3.4. Práctica P4: Entendiendo Inventor/VRML.

Objetivo: Aprender cómo es la estructura de una escena gráfica descrita por *Inventor* o *VRML*.

Material Disponible: Una escena sencilla que incluye varios nodos *"Shape, Property"* y *"Group"*.

Tarea: Abrir la escena usando *"Coindesigner2"* (Fig. 4) e interactivamente modificar los tres tipos de nodos mientras se visualiza: a) cambiar la geometría de un nodo *"Shape"*; b) cambiar los valores de una transformación y c) adicionar un nodo hijo a un nodo ya existente.

3.5. Práctica P5: Creando espacios virtuales con *Coin3D*.

Objetivo: Aprender cómo se visualiza una escena dentro de una interfaz hecha en *Qt*.

Material Disponible: El proyecto P3 en la que se ha aumentado la Interfaz con un visor 3D donde se pueden visualizar los objetos geométricos de la escena (Fig. 5).

Tarea: Completar las clases cone y cylinder imple-

```
<?xml version="1.0" encoding="UTF-8"?>
<Scene>
    <Geomobject>
        <File objectfile="objects/greencube.iv">
            Object File
        </File>
        <Type T="cube">
            Type of geomobject
        </Type>
        <Configuration TH="0.0" WZ="1.0" WY="0.0" WX="0.0" Z="0.0" Y="0.0" X="0.0">
            Object Position and Orientation
        </Configuration>
        <Properties p1="1.0" p2=" " p3=" ">
            p1 contains Cube side - p2 and p3 not used
        </Properties>
        <Scale sf="1.0">
            Scale factor to be applied to the ivfile
        </Scale>
    </Geomobject>

    . . .

    <Geomobject>
        <Type T="sphere">
            Type of geomobject
        </Type>
        <Configuration TH="0.0" WZ="1.0" WY="0.0" WX="0.0" Z="0.0" Y="1.0" X="0.0">
            Initial Configuration of the robot
        </Configuration>
        <Properties p1="0.5" p2=" " p3=" ">
            p1 contains Sphere radius - p2 and p3 not used
        </Properties>
    </Geomobject>
</Scene>
```

Figura 6: Definiendo los objetos de una escena a través de XML.

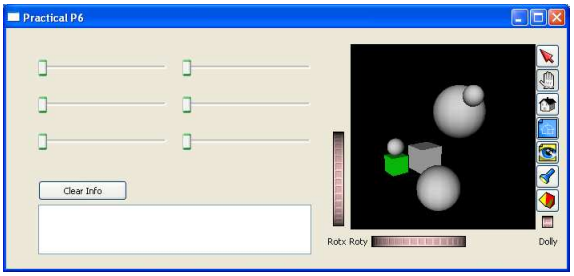

Figura 7: Interfaz de P6: Visualización de una escena definida por un archivo XML.

mentadas en la práctica P3 de forma que tengan las mismas capacidades gráficas.

3.6. Práctica P6: XML como mecanismo de entrada de datos.

Objetivo: Aprender cómo se describe una escena utilizando XML.

Material Disponible: El proyecto P5 a la que se ha adicionado una clase denominada structparse dedicada a cargar la escena descrita por un archivo XML. La Fig. 6 muestra parte de un archivo XML del proyecto que describe la escena de la Fig. 7 compuesta por cubos y esferas.

Tarea: Modificar la clase structparse de forma que pueda manejar correctamente los objetos de las nuevas clases cylinder y cone.

3.7. Práctica P7: Moviendo el robot: traslación y rotación de un objeto usando la librería *mt*.

Objetivo: Utilizar la librería *mt* para definir la posición y orientación de los objetos 3D, así como la interpolación de los movimientos entre diferentes configuraciones.

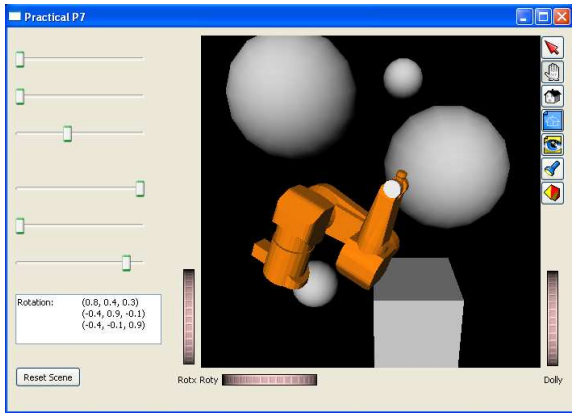

Figura 8: Interfaz de P7: Las barras de deslizamiento permiten cambiar la posición y orientación de un robot *"free-flying"* definido por una malla triangular.

Material Disponible: El proyecto P6 ampliado así: la clase geomobject ahora tiene como atributo una transformación definida por la clase Transform de la librería *mt*. La intefaz permite modificar la posición y la orientación de un objeto móvil e interpolar su movimiento desde la posición actual hasta la posición inicial a través del botón *"Reset Scene"* . Además, incluye una nueva clase mesh heredada de geomobject lo que permite manejar cualquier objeto tipo *"Shape"* descrito por un archivo *Inventor* (mallas triangulares, Fig. 8).

Tarea: Adicionar dos botones, uno para almacenar la configuración actual del objeto móvil y otro para interpolar el movimiento desde la configuración actual hasta la última almacenada.

3.8. Práctica P8: Verificando colisiones con PQP

Objetivo: Aprender cómo se realizan las consultas de colisión entre objetos que hace la librería *PQP* y cómo se obtienen los modelos *PQP* necesarios a partir de los modelos de *Inventor*.

Material Disponible: El proyecto P7 ampliado con la clase collisionobject y su clase derivada pqpobject (para hacer uso de otras librerías se pueden derivar su propias clases). Estas clases estan norganizasas en capetas separadas y se compilan como la librería libCollision. La clase geomobject contiene ahora un atributo de tipo puntero a un objeto de la clase collisionobject a través del cual se pueden realizar las consultas de collisión con los demás objetos de la escena (Fig. 9).

Tarea: Modificar la interfaz de forma que permita realizar consultas de colisión selectivas.

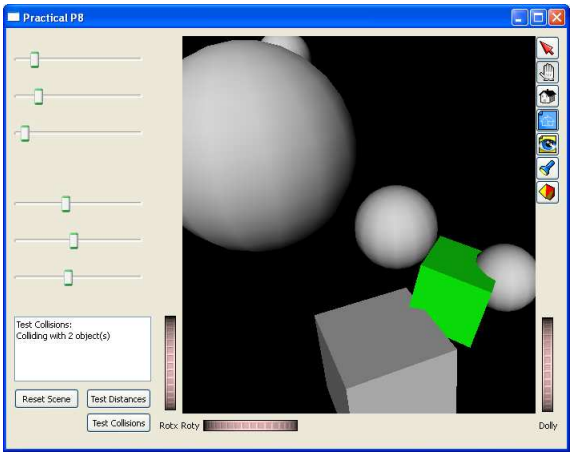

Figura 9: Interfaz de P8: la librería PQP permite realizar consultas de colisión o distancia entre objetos.

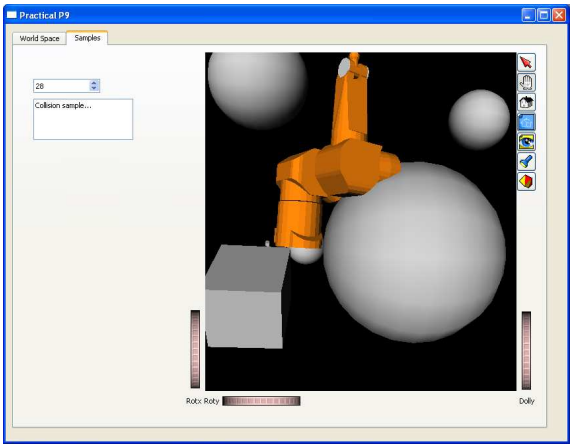

Figura 10: Interfaz de P9: La lengüeta activa muestra al robot *"free-flying"* en una de las configuraciones muestreadas e informa que es de colisión.

3.9. Práctica P9: Muestreando el *Cspace*

Objetivo: Centrar la atención en el modelados del los espacios de trabajo y de configuración, así como los procedimientos de muestreo.

Material Disponible: El proyecto P8 ampliado con las clases worldspace, cspace y sample. Estas clases están organizadas en carpetas separadas y se compilan como la librería libCspace. La interfaz esta organizada en dos lengüetas. La primera contiene la versión anterior a la que se han adicionado dos botones que realizan el muestreo. La segunda permite observar el objeto móvil en cada una de las configuraciones obtenidas del muestreo y conocer si es libre o de colisión (Fig. 10).

Tarea: Implementar un muestreo determínistico basado en la secuencia de Halton [5].

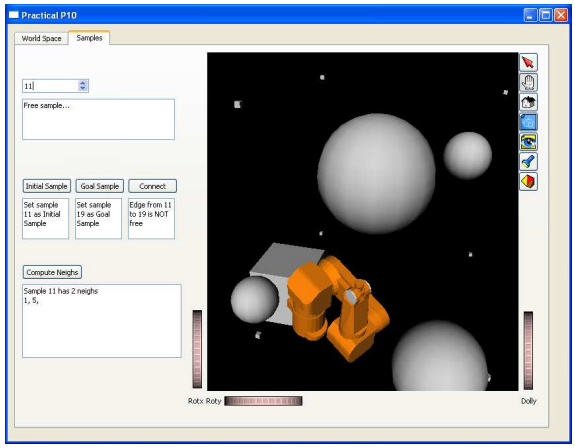

Figura 11: Interfaz de P10: Ésta permite probar si la arista que une dos muestras es libre o no y también calcular los vecinos de la muestra seleccionada.

3.10. Práctica P10: Búsqueda de vecinos y el planificador local

Objetivo: Hacer énfasis en dos necesidades básicas de los planificadores basados en *"Roadmaps"*: buscar el vecino más próximo e implementar un planificador local.

Material Disponible: El proyecto P9 ampliado con el algoritmo de búsqueda del vecino cercano (método de "fuerza bruta") y el planificador local implementado en la clase localplanner, que utiliza la estrategia incremental [14]. La interfaz permite definir de entre las muestras cuál será la configuración inicial y la final configuración y prueba si la arista que las úne es libre o no. También permite resolver consultas sobre los vecinos de una muestra (Fig. 11).

Tarea: Implementar el planificador local con el método binario [14]; como una práctica avanzada se puede implementar la búsqueda de vecinos cercanos con el algoritmo *ANN* [15].

3.11. Práctica P11: Manejo de grafos con la librería *Boost Graph*

Objetivo: Introducir la librería *"Boost Graph"* para realizar los procesos de búsqueda en grafos.

Material Disponible: Una versión simplificada del ejemplo A* proporcionado con la instalación de la librería.

Tarea: Modificar el ejemplo cargando diferentes grafos y cambiando la heurística utilizada.

3.12. Práctica P12: Mi primer PRM

Objetivo: Brindar una visión global de todas las partes involucradas en un planificador del tipo

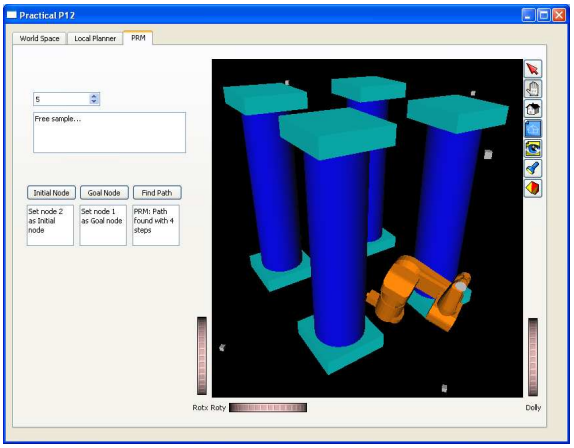

Figura 12: Interfaz de P12: El camino entre dos nodos del PRM puede ser calculado y visualizado.

"Roadmap" y las interacciones entre ellas.

Material Disponible: El proyecto P10 ampliado con la clase PRM que implementa el planificador utilizando las funcionalidades sobre grafos proporcionados por la librería *Boost Graph*. La Fig. 12 muestra una tercera lengüeta adicionada a la interfaz que permite al usuario seleccionar las configuraciones inicial y final y, si tiene solución, muestra al objeto moviéndose a lo largo de ella.

Tarea: Modificar el PRM introduciendo las opciones implementadas en las prácticas P9 y P10 (el muestreo determinístico y el método binario para el planificador local).

4. CONCLUSIONES

La creatividad necesaria para hacer aportaciones en cualquier campo de investigación debe estar sólidamente basada en el buen conocimiento de todos sus tópicos relacionados. Dentro del campo de la planificación de movimientos en robótica, debe tenerse en cuenta también todos los aspectos relacionados con la simulación. Este trabajo se ha escrito con el propósito de ayudar a los estudiantes de los cursos de robótica, en la rápida adquisición de habilidades con los aspectos más básicos relacionados con la planificación de trayectorias y la simulación de robots. Primero se propone un conjunto de herramientas multi-plataforma que siguen la filosofía de software libre. Luego, para suavizar la curva de aprendizaje de estas herramientas, se presenta un conjunto de doce prácticas que, de forma progresiva, muestran los conceptos básicos. Esta propuesta didáctica se está usando con éxito en el curso *"Planning in Robotics"* dentro del programa de doctorado *"Automatic Control, Robotics and Computer Vision"* de la Universitat Politècnica de Catalunya.

Referencias

[1] Arya, S., Mount, D. M., Netanyahu, N. S., Silverman, R., and Wu, A. Y. (1998) *J. ACM* **45(6)**, 891–923.

[2] Booch, G., Rumbaugh, J., and Jacobson, I. (1999) The Unified Modeling Language, Addison Wesley, .

[3] Booch, G. (2004) Object-Oriented Analysis and Design with Applications, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA 3rd edition.

[4] Collins-Sussman, B., Fitzpatrick, B., and Pilato, M. (2007) Control version with Subversion, O'Reilly, .

[5] Halton, J. H. December 1960 *Numerische Mathematik* **2(1)**, 84 – 90.

[6] Kavraki, L. and Latombe, J.-C. May 1994 In IEEE International Conference on Robotics and Automation (ICRA) volume **3**, : pp. 2138 – 2145.

[7] Kuffner, J.J., J. and LaValle, S. (2000) In IEEE International Conference on Robotics and Automation (ICRA) volume **2**, : pp. 995–1001.

[8] Larsen, E., Gottschalk, S., Lin, M. C., and Manocha, D. April 2000 In IEEE International Conference on Robotics and Automation (ICRA) : pp. 3719–3726.

[9] Lee, L.-Q., Siek, J. G., and Lumsdaine, A. (1999) In 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. New York: . pp. 399–414.

[10] Martin, K. and Hoffman., B. (2005) Mastering CMake: A Cross-Platform Build System, Kitware Inc., .

[11] Pérez, A. and Rosell, J. June 2008 *Computer Application in Engineering Education* **Accepted.**

[12] Rabin, S. (2003) AI Game Programming Wisdom 2, Charles River, .

[13] Rodríguez, A. The mt library Technical report June 2008.

[14] Roland Geraerts, M. H. O. (2006) *Robotics and Autonomous Systems* **54**, 165–173.

[15] Yershova, A. and LaValle, S. M. February 2007 *IEEE Transactions on Robotics* **23(1)**, 151–157.