POSICIONAMIENTO DE LAPBOT EN UN AMBIENTE TRIDIMENSIONAL VIRTUAL USANDO INTERFAZ HÁPTICA

SANDRA ANGELICA PUSIL ARCE
KELIN VICTORIA ZUÑIGA MENESES

Universidad del Cauca
Facultad de Ingeniería Electrónica y Telecomunicaciones
Departamento de Electrónica, Instrumentación y Control
Línea de I+D en Robótica y Control
Ingeniería en Automática Industrial

Popayán, Diciembre 2010

POSICIONAMIENTO DE LAPBOT EN UN AMBIENTE TRIDIMENSIONAL VIRTUAL USANDO INTERFAZ HÁPTICA

SANDRA ANGELICA PUSIL ARCE
KELIN VICTORIA ZUÑIGA MENESES

Tesis presentada a la Facultad de Ingeniería Electrónica y Telecomunicaciones de la Universidad del Cauca para la obtención del Título de

Ingeniero en Automática Industrial

Director:
Mag. Víctor Hugo Mosquera

Popayán, Diciembre 2010

Hoja de Aprobación

_____

_____

_____

Director _____
Mag. Víctor Hugo Mosquera

Jurado _____
Dr. Andrés Vivas Alban

Jurado _____
Dr. Carlos Gaviria López

Fecha de sustentación: Popayán, 19 de Enero de 2011

Agradecimientos

A Dios por acompañarnos en el transcurso de nuestros estudios de pregrado y permitirnos culminarnos con éxito.

A nuestros padres por su dedicación, apoyo y entrega incondicional.

A los ingenieros Sergio Salinas y Víctor Mosquera por su colaboración e interés para la realización de este proyecto como directores.

Al desarrollador de software Jorrit de Vries por su valiosa colaboración y grandes aportes al proyecto.

A la Ingeniera María Luisa Pinto por su interés y colaboración.

A nuestros compañeros de estudio por sus consejos, apoyo y amistad.

A los miembros del Departamento de Electrónica, Instrumentación y Control por su profesionalismo y cumplimiento en la labor de docentes.

A la Universidad del Cauca por su contribución administrativa y económica.

Resumen

En el mercado hay simuladores quirúrgicos como LAP Mentor, un simulador quirúrgico multi-disciplinario que permite la práctica de uno o varios alumnos, ó ProMIS que permite a los cirujanos entrenarse en modelos virtuales y físicos del cuerpo humano, en una misma unidad. Éstos son de gran utilidad para el entrenamiento de cirujanos, el problema es que tienen un costo elevado y muchas universidades no pueden acceder a ellos.

Este proyecto se constituye en la primera fase para el desarrollo de un simulador quirúrgico virtual, cuenta con un robot para laparoscopia (LapBot) que será posicionado a través de la interfaz háptica en el ambiente tridimensional, construido usando el programa de representación gráfica por computador Ogre3D.

Se usa el kit para desarrollo de software OgreHaptics, que permite comunicar la interfaz háptica con el motor de renderizado gráfico, para lograr que un objeto virtual creado en Ogre3D siga el movimiento generado por el usuario en el mundo real con la interfaz háptica Novint Falcon.

Se implementa el modelo geométrico inverso (MGI), en el lenguaje de programación C++, que permite calcular los valores de las variables θ_j (variable articular de rotación [rad]) y r_j (variable articular de traslación [m]) asociadas a las articulaciones, dependiendo de la orientación y localización deseada del efector final del robot en el espacio cartesiano (x, y, z).

Para aumentar la sensación de inmersión del usuario en el entorno se obtiene la detección de colisión haciendo uso del motor físico PhysX que utiliza un "*wrapper*" físico llamado NxOgre para ser usado con Ogre3D, y se genera una realimentación de fuerza al dispositivo, con lo cual el usuario puede sentir cuando el efector final del robot toca las paredes del abdomen simulado del paciente.

Palabras Clave: Simulador quirúrgico, interfaz háptica, laparoscopia, detección de colisiones.

Abstract

In the market there are surgical simulators as LAP Mentor, a multi-disciplinary surgical simulator that allows the practice of one or more students, or ProMIS that allows surgeons to train in virtual and physical models of the human body in one unit. These are useful for training surgeons, the problem is they are expensive and many universities can't access them.

This project constitutes the first phase of the development of a virtual surgical simulator that has a robot for laparoscopy (LapBot) to be positioned through the Novint Falcon Haptic Interface in a three-dimensional environment, built using the graphical representation computer program named Ogre3D.

Using the software development kit OgreHaptics, which allows communicating the haptic interface with graphic rendering engine to achieve that a virtual object created in Ogre3D follow the movement generated by the user in the real world with the haptic interface Novint Falcon.

Is implemented inverse geometric model (MGI), in the programming language C++, for calculating the values of the variables θj (rotation joint variable [rad]) and rj (translational joint variable [m]) associated with the joints, depending on the orientation and desired location of the end effector in cartesian space (x, y, z).

To increase the feeling of immersion to the user in the environment is performed collision detection using the PhysX physics engine through physics "*wrapper*" named NxOgre to use with Ogre3D, and to generate a force feedback to the device, thereby the user can feel when the end effector touches the walls of the simulated patient's abdomen.

Keywords: surgical simulator, haptic interface, laparoscopy, collision detection.

Contenido

Lista de Tablas

Lista de Figuras

Lista de Abreviaturas

3D:	Tridimensional.
API:	(*Application Programming Interface*) Interfaz de Programación de Aplicaciones.
CO_2:	Dióxido de Carbono.
GUI:	(*Graphical User Interface*) Interfaz Gráfica de Usuario.
LapBot:	Robot para Laparoscopia.
MGI:	Modelo Geométrico Inverso.
MIS:	(*Minimally Invasive Surgery*) Cirugía mini-invasiva.
OGRE:	(*Object-Oriented Graphics Rendering Engine*) Librería Orientada a Objetos de Representación Gráfica.
OpenGL:	(*Open Graphics Library*) Librería Gráfica de Libre Uso.
GPU:	Unidad de Procesamiento Grafico

Introducción

Hoy en día se utilizan robots para cirugía, los cuales permiten alcanzar una mayor precisión y velocidad en estos procedimientos, disminuyendo el esfuerzo físico del cirujano. En este tipo de robots se utilizan dispositivos hápticos que buscan aplicar el sentido del tacto a la interacción humana con sistemas informáticos. Las interfaces hápticas pueden incluir simultáneamente dispositivos de realimentación de fuerza y realimentación táctil. Las interfaces de realimentación de fuerza pueden ser vistas como extensiones del ordenador que aplican fuerzas físicas, y momentos sobre el usuario; las de realimentación táctil van a describir al usuario la superficie de contacto que se está recorriendo [1]. La importancia de estos dispositivos radica en que dan al usuario una sensación de realismo, factor de gran importancia para la práctica y ejecución de cualquier cirugía.

Los avances que ha sufrido el campo de la tecnología de la información y, dentro del mismo, la informática gráfica ha permitido el comienzo de la investigación y desarrollo de simuladores virtuales para el entrenamiento de especialistas. Una de las principales aplicaciones de estos avances es la creación de Simuladores Quirúrgicos. Este tipo de simuladores se caracteriza por sus grandes exigencias tanto visuales como del sentido del tacto, sentidos fundamentales en cirugía [1].

El uso de dispositivos automatizados en intervenciones médicas es un campo de rápida expansión que requiere de análisis de imágenes médicas, diseño y construcción de robots quirúrgicos y el desarrollo de interfaces hombre – máquina que mejore la relación entre el cirujano y estos sistemas. De los tipos de interacción hombre – máquina utilizada en la robótica médica, la realidad virtual aporta grandes ventajas para el desarrollo de simuladores de enseñanza, práctica y preparación de intervenciones [2].

Aunque ya se encuentran aplicaciones que incorporan herramientas de software y hardware para la creación de entornos virtuales quirúrgicos, aun falta aumentar las sensaciones de inmersión del cirujano en dicho entorno, que típicamente se dan de forma visual o auditiva, pero que pueden complementarse con información táctil para la percepción de tamaños, profundidades y dureza, etc., aumentando los niveles de realismo y confianza en estas aplicaciones [2].

Por esto la simulación virtual de una operación de cirugía es el resultado de combinar tres características fundamentales para alcanzar el realismo necesario que haga el sistema útil y convincente para el usuario. Debe lograrse:

1. Interacción en tiempo real con el usuario.
2. Una visualización realista de la simulación del interior del cuerpo humano.
3. Una realimentación de fuerza háptica de acuerdo a la interacción del cirujano con el cuerpo del paciente [3].

Actualmente, los grupos de investigación están dedicando grandes esfuerzos para alcanzar estos objetivos de realismo con el propósito de obtener una herramienta extremadamente potente que revolucione la docencia, la práctica y la difusión de las técnicas quirúrgicas [3].

En las últimas dos décadas ha sido planteada y explorada la realidad virtual como herramienta para la construcción de simuladores quirúrgicos más flexibles y económicos, sin embargo aun son muy costosos y están limitados sólo a unos cuantos procedimientos quirúrgicos [2].

Este proyecto permitirá desarrollar el primer simulador, en la Universidad del Cauca, de cirugía laparoscópica posicionado a través de una interfaz háptica con realimentación de contacto, lo que contribuye a la construcción de un primer entrenador virtual para cirujanos. Éste se diferencia de los existentes en que desarrolla un entrenador quirúrgico de bajo costo utilizando herramientas software de código abierto. Además el simulador contiene un robot virtual y utiliza la interfaz háptica Novint Falcon, adquirida por la Universidad en los últimos años.

A diferencia de otros trabajos de la Universidad, este proyecto implementa una interfaz háptica comercial de bajo costo en un proyecto de investigación en maestría terminado “Modelado, simulación en 3D y control de un robot para cirugía laparoscópica” [4].

En este proyecto se genera un sistema software para el posicionamiento del robot para cirugía laparoscópica (LapBot) en un ambiente virtual tridimensional (3D) usando la interfaz háptica Novint Falcon, logrando la realimentación de la colisión que realiza el efector final del robot sobre el abdomen simulado del paciente.

En el primer capítulo se presenta el estado del arte de los simuladores quirúrgicos y conceptos básicos sobre las interfaces hápticas, robótica quirúrgica y la cirugía mínimamente invasiva, que permiten exponer con mayor claridad el resto del documento. En el capítulo dos se describen los principales componentes del simulador y se muestra cómo se logró la comunicación entre Novint Falcon y LapBot, así como el posicionamiento del robot en el ambiente virtual utilizando este dispositivo háptico. El cómo se realizó la detección de colisiones entre el abdomen simulado del paciente y el efector final del robot para cirugía laparoscópica se puede observar en el capítulo tres, y por último se presentan las conclusiones del proyecto.

1. Simuladores Quirúrgicos

A continuación se presentan conceptos básicos sobre cirugía mínimamente invasiva, robótica quirúrgica, interfaces hápticas además de un resumido estado del arte sobre los simuladores quirúrgicos, su importancia en el entrenamiento de cirujanos, y el uso de la simulación tridimensional (3D) como una herramienta de apoyo.

1.1 Cirugía Mini – Invasiva (MIS)

Las cirugías abdominales tradicionalmente han supuesto largas incisiones, de tal forma que permitieran al cirujano tener un campo visual para realizar la operación. Sin embargo, este método ha conllevado una lenta recuperación de los pacientes [4].

Los avances en cirugía han creado nuevas técnicas y el material quirúrgico adecuado para evitar los inconvenientes debidos a grandes incisiones, dando lugar a la cirugía mini – invasiva [4].

La cirugía mini-invasiva es una técnica que permite intervenciones a través de incisiones muy pequeñas. Este procedimiento minimiza el trauma del paciente y requiere períodos de rehabilitación más cortos con respecto a la cirugía abierta. Sin embargo, debido a que la visibilidad y la operabilidad de las herramientas es reducida, constituye un procedimiento quirúrgico complejo [4].

Esta cirugía reduce el tamaño de la incisión y la estancia en hospital de tal forma que los pacientes pueden volver a su actividad normal en un periodo de tiempo más corto. Los nuevos dispositivos permiten a los cirujanos realizar muchas operaciones sin el trauma y el dolor de las primeras técnicas [4]. Algunas diferencias importantes entre la cirugía convencional y la MIS se resumen en la Tabla 1.1.

Tabla 1.1 Algunas diferencias entre cirugía convencional y MIS [4]

Cirugía Convencional	Cirugía Mini-Invasiva
Basada en la anatomía.	Basada también en simulación virtual.
Interfaz sensorial directa.	Interfaz video-endoscópica.
Depende de la habilidad del cirujano.	Depende también de la tecnología.
Gran cantidad de herramientas.	Cantidad limitada de herramientas.
Más tiempo de recuperación del paciente.	Menos tiempo de recuperación del paciente.
Grandes marcas estéticas.	Pequeñas marcas estéticas.

1.1.1 Laparoscopía

La cirugía laparoscópica es una técnica mediante la cual se realizan únicamente pequeñas incisiones de aproximadamente 1 cm. de diámetro sobre el abdomen del paciente. A diferencia de la cirugía tradicional, en lugar de mirar directamente al órgano del cuerpo que está siendo tratado, los médicos monitorizan el procedimiento a través de una videocámara especial llamada laparoscópio. Esta cámara se introduce a través de uno de los pequeños agujeros, mientras los otros instrumentos son manipulados desde los otros puntos de incisión [4].

Contrariamente a la cirugía “abierta” donde los cirujanos pueden manipular y palpar (tocar) el tejido u órgano durante la operación, cualquier procedimiento efectuado en la laparoscopia es llevado a cabo utilizando un instrumento laparoscópico diseñado para encajar dentro del cuerpo a través de unas guías llamadas trocar, que entra a través de la pequeña incisión. El objetivo de este diseño es transformar el movimiento de la mano del cirujano a través de un largo tubo de pequeño diámetro creando una o más funciones de salida en la punta opuesta, dentro de la cavidad del cuerpo [5].

En laparoscopia es necesario separar la pared abdominal de los órganos, elevando el abdomen, para que el cirujano tenga acceso al interior del paciente y pueda manipular los instrumentos con relativa facilidad. La elevación se realizó en un principio, con insuflación por aire, después se usó óxido nitroso y actualmente se emplea CO_2, dado que suprime la combustión y el cuerpo lo absorbe con mayor rapidez [6].

Por medio de los trocares se insertan pinzas, tijeras, cauterizadores o cualquier otro instrumento que se requiera, además del sistema de visión (endoscopio) [4]. La Figura 1.1 muestra la configuración de los elementos, los trocares aparecen de color violeta y el sistema de visión posee el cable de luz óptica.

Figura 1.1 Configuración de elementos básicos para laparoscopia[1].

La cirugía laparoscópica puede ser clasificada como funcional o ablativa. La primera incluye procedimientos reparativos y la segunda supone la extirpación de órganos o algunas de sus partes. Actualmente, los procedimientos quirúrgicos laparoscópicos más comunes son [7]:

- **Colecistectomía**: intervención que se realiza para quitar una vesícula biliar enferma, que está inflamada u obstruida por cálculos biliares.

- **Apendicectomía:** extirpación de un apéndice inflamado o infectado (apendicitis). Se hace una incisión pequeña en el cuadrante inferior derecho del abdomen y se extirpa el apéndice.

- **Histerectomía**: cirugía para extirpar el útero (matriz). El cirujano realiza una incisión por la cual corta las trompas de Falopio y separa el útero en su unión con el cuello.

- **Colectomía asistida**: extirpación o resección de una parte enferma del intestino grueso (colon).

- **Ligadura de trompas**: es una forma permanente de evitar el embarazo cerrando las trompas de Falopio de una mujer.

- **Gastrectomía**: extirpación parcial o total del estómago, en caso de presentarse problemas gástricos crónicos como úlceras o cáncer.

- **Prostatectomía:** intervención para extraer la totalidad o parte de la glándula prostática (próstata), para la curación del cáncer, la preservación de la continencia o la función sexual.

[1] www.versatius.com

- **Hepatectomía**: procedimiento para diagnosticar, tratar o evacuar las lesiones y tumores en el hígado.

- **Vasectomía**: procedimiento para lograr la esterilización masculina. Consiste en cortar y ligar los conductos deferentes, impidiendo el recorrido del semen.

1.2 Robótica Quirúrgica

El procedimiento llamado laparoscopia, se traduce en menores traumas para el paciente y consecuentemente, en una recuperación más rápida, así como menores costos para el sistema de seguridad social. Sin embargo trajo nuevos retos y dificultades para el cirujano como la pobre realimentación en las sensaciones al tocar los órganos internos, la pérdida de la visión en tres dimensiones, la pérdida de la articulación de la muñeca y la pobre ergonomía de los instrumentos utilizados. Se puede resumir diciendo que el paciente ha ganado mucho a expensas del mayor trabajo realizado por el cirujano. Surgió entonces la idea de utilizar robots de asistencia para eliminar estos nuevos impedimentos, añadiéndole además la mejora en el trabajo a escala a realizar y filtrando el temblor del operador humano. Inclusive algunos autores han sugerido que en la historia de la evolución quirúrgica, la laparoscopia no es más que una transición tecnológica que llevaría desde los procedimientos manuales hacia la cirugía robotizada [8].

La robótica quirúrgica ha avanzado rápidamente, desde que en 1985 se utilizó un simple robot Puma industrial para realizar un procedimiento neuroquirúrgico, consistente en la introducción de una guía para realizar una biopsia cerebral. Dado que el robot utilizado fue un simple robot industrial el cual no cumplía con las condiciones de seguridad impuestas en un bloque operatorio, se empezaron a diseñar los primeros robots con propósitos puramente quirúrgicos. Los pioneros en este nuevo campo fueron un robot para cirugías de próstata desarrollado en el Reino Unido, y un robot para neurocirugía desarrollado en Francia. Desde entonces se han realizado miles de procedimientos quirúrgicos con cientos de robots instalados en diferentes hospitales del mundo, los cuales asisten en operaciones de cardiología, urología, otorrinolaringología, neurocirugía, ortopedia, cirugía maxilofacial, radiocirugía, oftalmología y cirugía abdominal [8]. Entre los robots quirúrgicos más populares e interesantes podemos contar:

- **Aesop**

Fue el primer robot utilizado en aplicaciones quirúrgicas comercializado en los Estados Unidos, en el año 1994. El robot Aesop mueve un endoscopio (cámara de video) al interior del cuerpo humano en operaciones de laparoscopia, y es controlado por la voz del cirujano.
Esto le permite al médico una mayor libertad de movimientos, centrando su atención en el área deseada a partir de las órdenes vocales dadas al robot. Sin embargo la tarea desarrollada por el robot no involucra una activa manipulación invasiva al interior del cuerpo humano [8].

- **Zeus y Da Vinci**

Se trata de dos sistemas teleoperados con características comunes: el cirujano está cómodamente sentado frente a una consola, observando en una pantalla una imagen tridimensional del paciente, y manejando dos instrumentos maestros hápticos que mueven dos instrumentos quirúrgicos esclavos ubicados al interior del paciente. El robot Zeus consta de seis grados de libertad mientras que el sistema Da Vinci consta de siete. Es de anotar que las respectivas compañías fabricantes de estos dos sistemas se fusionaron en el año 2003 y el sistema que se sigue comercializando es el Da Vinci [8].

1.3 Simulación en 3D

En el proceso de aprendizaje de un cirujano se precisa tarde o temprano de la intervención sobre un paciente. Éste puede estar vivo o no, puede ser humano o algún animal o incluso un maniquí, pero en cualquiera de los casos es un ensayo destructivo que no se puede repetir con el mismo paciente en las mismas circunstancias. Por lo tanto sólo se pueden entrenar algunas patologías, las que se encuentren en el paciente o simulados en un maniquí [3].

La creación de entrenadores de realidad virtual para cirugía permite repetir la misma operación tantas veces como se quiera sin tener que depender de un paciente físico para el aprendizaje de una determinada técnica [3].

Una aplicación típica es el entrenamiento en Laparoscopía, del que existen varios modelos, algunos de ellos se muestran a continuación.

- **LAP MENTOR**

El LAP Mentor (Figura 1.2) es un simulador quirúrgico multi-disciplinario, permite la práctica de uno o varios alumnos. El sistema ofrece oportunidades de capacitación a los cirujanos nuevos y experimentados, desde el perfeccionamiento de habilidades básicas de laparoscopía hasta realizar procedimientos quirúrgicos laparoscópicos completos [9].

Figura 1.2 Lap Mentor [9].

- **Simulador quirúrgico: The LaparoscopyVR Virtual-Reality System**

El simulador quirúrgico LapVR (Figura 1.3) utiliza los últimos avances en tecnología, adecuados modelos en 3D interactivos, realimentación de fuerza táctil, seguimiento y evaluación del desempeño para ayudar a disminuir la curva de aprendizaje en cirugía laparoscópica [10].

Proporciona una variedad de casos y escenarios de entrenamiento reales de formación, con el aumento de niveles de dificultad que permiten a los médicos adquirir práctica, repetir y mejorar sus habilidades antes de siquiera tocar un verdadero paciente. Una amplia gama de fuerzas puede ser experimentada, como los tejidos responden a la manipulación de las herramientas, las fuerzas pueden ser ajustadas utilizando el sintetizador háptico incluido [10].

Figura 1.3. Laparoscopy VR [10].

- **Virtual Laparoscopy Simulator (LapSim)**

Este sistema digital de simulación laparoscópica (Figura 1.4) constituye una experiencia de aprendizaje efectiva permitiendo a los estudiantes practicar las habilidades básicas de la cirugía laparoscópica en un ambiente de realidad virtual [11].

El entrenamiento de las habilidades básicas incluye la navegación con cámara, la navegación con instrumental, la coordinación, grapado, disección, colocación de clips, sutura y medición de precisión y velocidad. En todos los ejercicios el cirujano debe identificar el objeto propuesto y ubicar los instrumentos, realizando la tarea requerida con la mayor precisión y en el menor tiempo posible [11].

Figura 1.4 LapSim [11].

- **ProMIS**

Permite a los cirujanos entrenarse en modelos virtuales y físicos del cuerpo humano, en una misma unidad. Utiliza instrumentos y materiales quirúrgicos reales. Dispone de aparatos de lectura gráfica y vídeo. El simulador ofrece un modelo físico en el que pueden entrenarse en la colocación de trocares y el uso de instrumental. Los instrumentos pueden ser intercambiados, retirados y reinsertados. Las señales de audio contribuyen a que la experiencia sea más realista. Los cirujanos también pueden entrenar en equipo. Cada módulo de aprendizaje de ProMIS (Figura 1.5) se divide en distintas tareas y niveles, lo que proporciona una trayectoria clara en el aprendizaje del usuario. ProMIS analiza, entre otras cosas, el tiempo empleado, las longitudes de las trayectorias o la suavidad de los movimientos del cirujano: en qué grado sus movimientos son fluidos o erráticos. El plan de entrenamiento básico de ProMIS incluye: orientación del laparoscópio, dirección del instrumental, disección, diatermia y suturas [12].

Figura 1.5 ProMIs [13].

- **Universidad Politécnica de Valencia: Simulador para el entrenamiento en cirugías avanzadas**

Es un simulador de laparoscopia virtual (Figura 1.6), desarrollado por el grupo de investigación MediCLab, el cual permite familiarizar a los cirujanos tanto con el manejo de las herramientas como en las diferentes técnicas de intervención en este tipo de cirugías. Este simulador posee dos ventajas frente a los sistemas de entrenamiento tradicionales: permite repetir una técnica tantas veces como sea necesario hasta su correcto aprendizaje y permite simular la intervención sobre la reconstrucción 3D de los órganos del paciente previamente a su intervención quirúrgica real [14].

Figura 1.6 Simulador desarrollado por MediCLab [14].

- **THUMP: Una consola Háptica de Inmersión para Simulación y Entrenamiento Quirúrgico**

La consola háptica THUMP incorpora gafas estereoscópicas y dos mecanismos maestros de ocho ejes. En la consola, el operador puede ver imágenes reales o virtuales de la cirugía a través de las gafas, proporcionando una inmersión tridimensional en el entorno. El operador sostiene los dos efectores de los mecanismos maestros entre el pulgar y el dedo índice, tal como se muestra en la Figura 1.7, permitiendo al sistema medir la posición, orientación y el nivel de adherencia de las dos manos. Un computador, con software RTAI Linux, ejecuta los comandos de control necesarios para aplicar las fuerzas. La simulación de los mecanismos maestros permite la predicción de las colisiones y detección de errores [15].

Figura 1.7 Consola THUMP [15].

- **Universidad Central de Venezuela: Laparos**

Laparos (Figura 1.8) es un sistema de realidad virtual para entrenamiento en cirugía laparoscópica. El componente hardware está conformado principalmente por un simulador mecánico y un subsistema de rastreo, El componente software está conformado por un subsistema de información y uno gráfico tridimensional [16].

Las sesiones de entrenamiento para estudiantes de cirugía laparoscópica, incluyen modelos deformables para simular la deformación de los tejidos producida por una acción específica, detección de colisiones para proporcionar una retroalimentación visual cuando, por ejemplo, un instrumento toca un órgano virtual [16].

Figura 1.8 Laparos [16].

- **Universidad EAFIT "Simulador de Cirugías mínimamente invasivas"**

En este proyecto se desarrolló la primera fase de un simulador de cirugía mínimamente invasiva (Figura 1.9) basado en realidad virtual, para apoyar la formación de cirujanos en los procedimientos quirúrgicos. Dentro de los procedimientos que puede realizar el cirujano en el simulador está la movilidad de los instrumentos y del laparoscópio, manipulación y agarre de estructuras anatómicas. Para cumplir con estas tareas se reconstruyó tridimensionalmente el hígado a partir de las imágenes del proyecto "Visible Human" y se creó un modelo tridimensional de los instrumentos laparoscópicos, se implementó en la GPU[2] un modelo físico que simulara la deformación del órgano al contacto con los instrumentos, también se desarrolló un algoritmo de detección de colisiones para determinar cuando el objeto está en contacto con el órgano y finalmente se diseñó y construyó una interfaz física que simulara las condiciones reales de la cirugía, y así el cirujano pudiera interactuar de una forma intuitiva con el ambiente virtual. A dicha interfaz física se le integraron dos dispositivos de retroalimentación con el fin de adicionar una percepción de fuerza del ambiente virtual [1].

Como trabajos futuros del simulador el cual se acondicionará como herramienta educativa, se desarrollarán los módulos faltantes correspondientes a la simulación de procedimientos básicos quirúrgicos, como corte y sutura, un módulo pedagógico. Adicionalmente se explorarán modalidades colaborativas con el fin de soportar el aprendizaje remoto de habilidades quirúrgicas [1].

[2] GPU: Unidad de Procesamiento Gráfico

Figura 1.9 Simulador quirúrgico EAFIT [1].

- **Desarrollo de un sistema de entrenamiento médico para artroscopia de rodilla**

Se presenta el desarrollo de un prototipo de simulador (Figura 1.10) para entrenamiento en los procedimientos médicos de artroscopia de rodilla desarrollado por la Universidad de los Andes. El objetivo es implementar un sistema que permita simular algunos procedimientos artroscópicos empleando un dispositivo háptico de retorno de fuerza. Para esto se construyó un modelo tridimensional de la articulación de la rodilla. Con base en imágenes de resonancia magnética, se simulan algunos de los efectos percibidos durante una artroscopia real y se implementan las escenas gráficas y hápticas que permitan contrastar las sensaciones que se perciben en un proceso artroscópico [17].

Figura 1.10 Prototipo de simulador de artroscopia operando [17].

1.4 Interfaces Hápticas

El término Háptico proviene de un término griego que significa “capaz de coger o asir” .La morfología del término está en las raíces hap·tic ('hap-tik) del griego haptiko adjetivo asociado al sentido del tacto haptesthai [3].

Con el término “interfaces hápticas” se alude a aquellos dispositivos que permiten al usuario tocar, sentir o manipular objetos simulados en entornos virtuales y sistemas teleoperados. En la mayoría de simulaciones realizadas en entornos virtuales, basta con emplear displays 3D y dispositivos de sonido 3D stereo para

provocar en el usuario, mediante imágenes y sonidos, la sensación de inmersión dentro del espacio virtual. No obstante, además de provocar en el usuario esta sensación de inmersión, se debe proporcionar la posibilidad de interactuar con el medio virtual, pudiendo establecer entre el usuario y el entorno virtual una transferencia bidireccional y en tiempo real de información mediante el empleo de interfaces de tipo háptico [18].

Algunos de los principales campos de aplicación de las interfaces hápticas son [18]:

- Medicina: Simuladores quirúrgicos para entrenamiento médico, micro robots para cirugía mínimamente invasiva (MIS), etc.

- Educacional: Proporcionando a los estudiantes la posibilidad de experimentar fenómenos a escalas nano y macro, escalas astronómicas, como entrenamiento para técnicos, etc.

- Entretenimiento: Juegos de video y simuladores que permiten al usuario sentir y manipular objetos virtuales, etc.

- Industria: Integración de interfaces hápticos en los sistemas CAD de tal forma que el usuario puede manipular libremente los componentes de un conjunto en un entorno inmersivo.

- Artes gráficas: Exhibiciones virtuales de arte, museos, escultura virtual etc.

Las interfaces hápticas en el campo de la cirugía ofrecen una percepción táctil de órganos y tejidos virtuales. Añadido a la visualización de los mismos, concede un mayor grado de realismo a un entorno de trabajo ficticio.

Gracias al avance producido tanto en el campo de la robótica como en los ordenadores, se permite el desarrollo de aplicaciones de simulación virtual pudiendo tener múltiples aplicaciones dentro del ámbito de la docencia quirúrgica.

Mediante los simuladores quirúrgicos se desarrollan disciplinas de entrenamiento para cirujanos, además de ampliar la capacidad de percibir sensaciones táctiles virtuales. Esto genera un amplio espectro de posibilidades tanto en el área del entrenamiento como en la preplanificación de intervenciones, siendo muy interesantes cuanto más complejas sean las operaciones [3].

A continuación se muestran algunas de las interfaces hápticas utilizadas actualmente:

1.4.1 Interfaces Hápticas Desktop Con Feedback De Fuerza

- **PHANTOM Sensable Techologies**

Actualmente se dispone de varios modelos de esta interfaz (Figura 1.11), cuyo número de grados de libertad en posicionamiento varía desde 3 hasta 6, pudiendo recibir *forcefeedback* a lo largo de todos o algunos de estos grados de libertad. El espacio de trabajo de los distintos modelos varía considerablemente desde los modelos iniciales a los superiores. La fuerza máxima que puede proporcionar es de 22N en el modelo Premium 3.0 y la fuerza sostenida (24h.) es de 3N. Conforme avanzamos hacia modelos superiores, aumenta la inercia de los dispositivos, al tiempo que disminuye su rigidez. El Phantom renueva el estado de sus fuerzas cada milisegundo, y presenta una alta resolución posicional [18].

Figura 1.11 Phantom Sensable Technologies [18].

En la Tabla 1.2 se pueden apreciar las especificaciones técnicas de esta interfaz háptica.

Tabla 1.2 Especificaciones técnicas de Phantom [18]

Force feedback workspace	~6.4 W x 4.8 H x 2.8 D in. > 160 W x 120 H x 70 D mm.
Footprint (Physical area device base occupies on desk)	6 5/8 W x 8 D in. ~168 W x 203 D mm.
Weight (device only)	3 lbs. 15 oz.
Range of motion	Hand movement pivoting at wrist
Nominal position resolution	> 450 dpi. ~ 0.055 mm.
Backdrive friction	< 1 oz (0.26 N)
Maximum exertable force at nominal (orthogonal arms)position	0.75 lbf. (3.3 N)
Continuous exertable force (24 hrs.)	> 0.2 lbf. (0.88 N)
Stiffness	X axis > 7.3 lbs. / in. (1.26 N / mm.) Y axis > 13.4 lbs. / in. (2.31 N / mm.) Z axis > 5.9 lbs. / in. (1.02 N / mm.)
Inertia (apparent mass at tip)	~0.101 lbm. (45 g)
Force feedback	x, y, z
Position sensing [Stylus gimbal]	x, y, z (digital encoders) [Pitch, roll, yaw (± 5% linearity potentiometers)
Interface	IEEE-1394 FireWire® port: 6-pin to 6-pin
Supported platforms	Intel or AMD-based PCs
OpenHaptics® Toolkit compatibility	Yes

Applications	Selected Types of Haptic Research, FreeForm® Modeling™ system, ClayTools™ system

- **FREEDOM 6S MPB Technologies**

Esta interfaz (Figura 1.12) posee 6 GDL y un nivel de fricción de aproximadamente 0.1 N en cada dirección. La inercia resultante en el extremo varía entre 0.09 y 0.15 Kg [18].

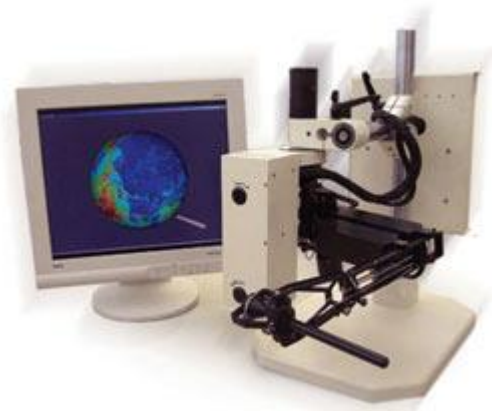

Figura 1.12 Freedom 6S MPB

En la Tabla 1.3 se pueden observar las especificaciones técnicas de este dispositivo háptico.

Tabla 1.3 Especificaciones técnicas de Freedom 6S MPB [18]

Environmental				
Mounting			Table top unit	
Footprint			25 x 25 cm	
Device Operating Temperature			22 ± 5 °C	
Mass of device with stand			(45.3 lb)	
Mass of device without stand			(11.2 lb)	
Control				
Connection			Two PCI or PCIe cards installed in a PC, with cable connection to device	
Operating System			Windows, Linux	
Update Rate 1kHz recommended				
Mechanical	Translation	Pitch	Yaw	Roll
Workspace	17x22x33 cm	170°	130°	340°
Tip Inertia	125 g	0.04 g.m2	0.03 g.m2	0.003 g.m2
Backdrive Friction	40 mN	5 mN.m	2 mN.m	1 mN.m
Max Force/ Torque	2.5 N* 0.6 N***	370 mN.m** 105 mN.m***	310 mN.m** 88 mN.m***	150 mN.m** 44 mN.m***
Position Resolution	2 µm	0.02 mrad	0.02 mrad	0.02 mrad
Force Resolution	1.5 mN	0.1 mN.m	0.1 mN.m	0.05 mN.m
Stiffness	2 N/mm	4.0 N.m/rad	2.5 N.m/rad	0.2N.m/rad
Electrical				

- **NOVINT TECHNOLOGIES**

Novint Falcon (Figura 1.13) es un dispositivo fabricado por Novint Technologies, Inc. [19] para interacción en escenarios 3D, con realimentación de fuerzas. Se ha diseñado inicialmente para aplicaciones de entretenimiento, videojuegos o sustitución de periféricos como el ratón o el joystick y por su bajo costo se ha convertido en el pionero en la categoría de productos hápticos para el mercado de consumo.

Figura 1.13 Novint Falcon [19]

2. Posicionamiento de LapBot Usando la Interfaz Háptica Novint Falcon

En este capítulo se describen los principales componentes del simulador. Además se muestra cómo se logró la comunicación entre la interfaz háptica Novint Falcon y el ambiente virtual 3D, donde se encuentra el robot, y cómo se realizó el posicionamiento de LapBot usando este dispositivo háptico.

2.1 Principales Componentes del Simulador

2.1.1 LapBot: Robot Para Cirugía Laparoscópica

LapBot es un robot para cirugía laparoscópica que presenta una estructura de nueve grados de libertad, con seis articulaciones activas que permiten posicionar y orientar el instrumento quirúrgico en un espacio tridimensional; y tres pasivas, también muy importantes, que permiten mantener el punto fijo de movimiento sobre el punto de incisión donde se encuentra el trocar. Con estos 3 grados de libertad se garantiza que el robot es intrínsecamente seguro y que no dañará al paciente por un esfuerzo agresivo en la incisión [4].

En la Figura 2.1 se puede observar la estructura cinemática para LapBot, la articulación de traslación se representa con un prisma y las de rotación con cilindros, cada una con su respectivo número.

El punto de incisión por donde cruza el trocar en la cirugía se ha representado con un anillo que se encuentra entre las articulaciones 7 y 8, de tal forma que sólo dos articulaciones quedan dentro de la cavidad abdominal del paciente [4].

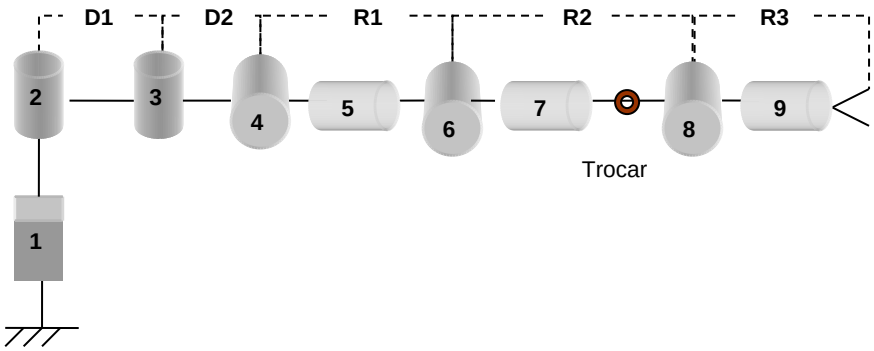

Figura 2.1 Estructura cinemática del robot LapBot [4].

Las tres primeras articulaciones permiten posicionar a LapBot en un espacio tridimensional, la cuarta articulación se fija en un valor determinado para inclinar al robot y ubicarlo por encima del abdomen del paciente, las articulaciones quinta y sexta son pasivas y mantienen el movimiento del cuerpo de longitud *R2* a través del trocar, y las últimas tres articulaciones forman una estructura en forma de muñeca que permite la orientación de la herramienta [4].

Las distancias del robot se han etiquetado con letras *D* y *R* de dimensiones: *R3*=0.01 m. (tamaño de la punta de la pinza), *R2*=0.3 m. de longitud y 0.008 m. de diámetro (cuerpo que pasa por el trocar). Como el cuerpo de longitud *R1* es aquel que permite posicionar al robot por encima del abdomen del paciente, para que el resto de los cuerpos no colisionen, la mínima dimensión de *R1* será de 0.2 m [4].

Ahora, teniendo en cuenta que se colocarán dos brazos de LapBot a lado y lado de la camilla y a no más de 0.05 m. de su orilla, y suponiendo que la incisión al paciente no se realizará a más de 0.3 m. de la base del robot, la posición que se presenta en la Figura 2.2 es la que exige la mayor elongación de LapBot. Como se observa, el cuerpo *R2* queda casi totalmente retraído lo cual indica que la suma de *D1, D2* y *R1* es aproximadamente 0.65 m. Como *R1* = 0.2 m., entonces *D1+D2* = 0.45 m. Luego, para darle mayor versatilidad y espacio de trabajo sobre la camilla se escoge *D1* = 0.25 m. y *D2* = 0.20 m. [4].

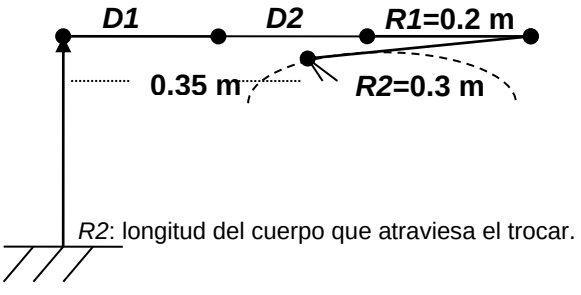

Figura 2.2 Máxima elongación para LapBot [4].

2.1.2 Estructura del abdomen simulado

El modelo matemático del cuerpo humano desarrollado por Hanavan determina que el abdomen de un hombre adulto promedio de 1.60 m de estatura y 60 Kg. de peso, es similar a un paralelepípedo de 0.164 m. x 0.224 m. x 0.274 m. [4].

También se tiene en cuenta los 0.15 m. que se eleva el abdomen debido al gas de CO_2 [4]. La Figura 2.3 muestra un esquema aproximado del abdomen insuflado y las ubicaciones de las dos incisiones realizadas para introducir los instrumentos quirúrgicos.

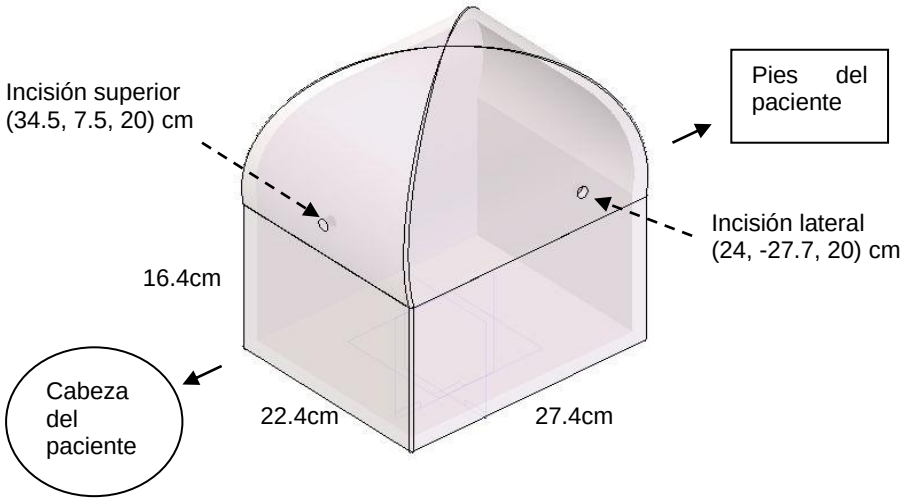

Figura 2.3 Modelo estilo caja del abdomen del paciente [4].

Las coordenadas de los puntos de incisión se calculan a partir de las bases de los robots que se colocan en los bordes de la camilla, en las mismas ubicaciones donde se encuentran el cirujano y el asistente [4].

2.1.3 Interfaz Háptica Novint Falcon Technologies

Novint Falcon es un dispositivo háptico de tipo comercial fabricado por Novint Technologies, consiste en un mango conectado a tres brazos, y éstos a un cuerpo en forma cónica que descansa en una base en forma de U. Cada uno de los brazos se puede mover hacia dentro y hacia afuera del cuerpo del Falcon. El mango es una pequeña esfera con cuatro botones. En la parte frontal del dispositivo se pueden observar unos LEDs que indican el estado del mismo. El cuerpo contiene tres motores, cada uno conectado a los brazos de la interfaz. Mientras los tres brazos se mueven un sensor óptico esta enlazado a cada motor para seguir los movimientos del brazo. Una matriz Jacobiana es usada para determinar la posición del cursor 3D en coordenadas cartesianas basándose en la

posición de los brazos. La posición de este cursor háptico es por lo tanto controlada por los movimientos de la interfaz y es usada por el software del Falcon para determinar las fuerzas que van a ser aplicadas al dispositivo. Las corrientes son enviadas a los motores a una tasa de 1KHz para presentar al usuario un sentido de tacto real. De esta forma una fuerza puede ser aplicada al mango en cualquier dirección, de cualquier magnitud, alrededor de 2 lbs. de fuerza cada 0.1 milisegundos.

Esencialmente, se trata de un joystick 3D que responde con toda la gama de fuerza: peso, forma, textura, dimensión, dinámica y efectos de la fuerza. Esto permite que el usuario experimente una amplia gama de sensaciones táctiles reales.

El Novint está destinado a traducir la sensación de peso e inercia al sostenerlo, así que si el avatar[3] del jugador está sosteniendo una pelota, el Novint arrastrará hacia abajo con el peso de la pelota.

Por otro lado, Novint Falcon tiene un costo entre $ 100 - $ 300 us lo cual disminuye drásticamente el costo de los sistemas hápticos, en el rango de precio razonable para un usuario medio.

Este dispositivo háptico mostrado de la
Figura 2.4, es un esclavo de arquitectura paralela, con un espacio de trabajo de 4" x 4" x 4", capaz de soportar fuerzas mayores a 2 lbs. y una resolución de posición mayor a 400 dpi, otras especificaciones se presentan en la Tabla 2.1 y en la Tabla 2.2 [2].

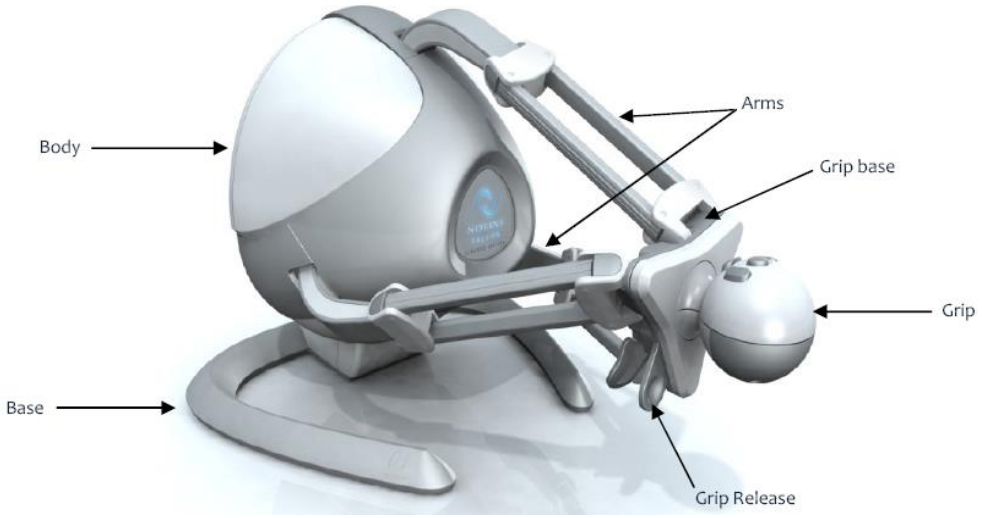

Figura 2.4 Novint Falcon [19].

[3] En un ambiente tridimensional el Avatar es la representación digital de quien está visitando el ambiente.

Tabla 2.1 Especificaciones Técnicas Del Dispositivo Háptico Novint Falcon [2].

NOVINT FALCON ™	
Hardware Specifications	
Size	9" x 9" x 9"
Weight	6 lbs.
Position Resolution	> 400 dpi
Quick Disconnect Handle	< 1 second change time
Communication Interface	USB 2.0
Size	9" x 9" x 9"
Power	30 watts, 100V- 240V, 50Hz-60Hz
Degrees of Freedom	3DOF input, 3DOF output Additional DOF possible through enabled grips
Software Specifications	
Supported Platforms	Launch platform – PC, on Windows XP and Vista Anticipate migration to PS3 and XBox 360
Minimum System Requirements	Processor: 1.0 GHz Processor **Graphic card: 128Mb 3D hardware accelerated graphics card.** DirectX Version: DirectX 9.0c Hard Drive: 1.5 GB free disk space Memory: 512 MB RAM Other: DVD-ROM Drive (CDs available upon request), USB 2.0 connection
API	• C++ SDK overview • API layers – DHDLC (low-level drivers) – HDAL (mid-level device communication) – Falcon API (high-level programming toolset) • Allows for integration into existing games and virtually every existing game engine
Haptic Specifications	
3D Touch Workspace	4" x 4" x 4"
Force Capabilities	> 2 lbs 8.896N
Separate haptics and graphics loops Single point interaction. Must Maintain device stability Haptics wall (F = -kx) 3DOF with feedback forces	

Tabla 2.2 Especificaciones Entradas y Salidas del Sistema Novint Falcon [2].

ENTRADAS	
DIGITALES	ANALOGAS
4: Botones 1, 2, 3 y 4	3: Encoder 1, 2 y 3
SALIDAS	
DIGITALES	ANALOGAS
3: Leds 1, 2 y 3	3: Motores 1, 2 y 3

2.1.4 Motor de Simulación 3D

El motor de simulación 3D es Ogre3D (*Object-Oriented Graphics Rendering Engine*). Ogre es un motor gráfico de fuente abierta orientado a objetos escrito en C++. Tiene licencia GNU Licencia Pública General Menor (LGPL) que permite su uso libre con algunas pequeñas restricciones. Fue diseñado con el objetivo de facilitar el trabajo de los desarrolladores que producen aplicaciones basadas en hardware de aceleración gráfica 3D [20].

Desde 2001 OGRE ha crecido hasta convertirse en uno de los más populares motores de renderizado gráfico de fuente abierta, y se ha utilizado en un gran número de proyectos, en áreas tan diversas como juegos, simuladores, software educativo, arte interactivo, visualización científica, y otros [21].

Utiliza librerías gráficas OpenGL y Direct3D de Microsoft, lo que permite la portabilidad del código a diferentes plataformas como Linux, Windows, Mac OS. Posee una comunidad altamente activa que a través de su foro participa desarrollando *plugins* que son capaces de integrar a Ogre con una gran cantidad de herramientas gráficas y da soporte continuo tanto a iniciados como a desarrolladores expertos. Lo que lo convierte sin duda en una atractiva opción para integrar al simulador.

Ogre tiene una gran cantidad de clases y subclases que le permiten realizar su trabajo. Para entender a grandes rasgos como trabaja Ogre es útil el siguiente diagrama UML (Figura 2.5) que incorpora las clases más importantes [20].

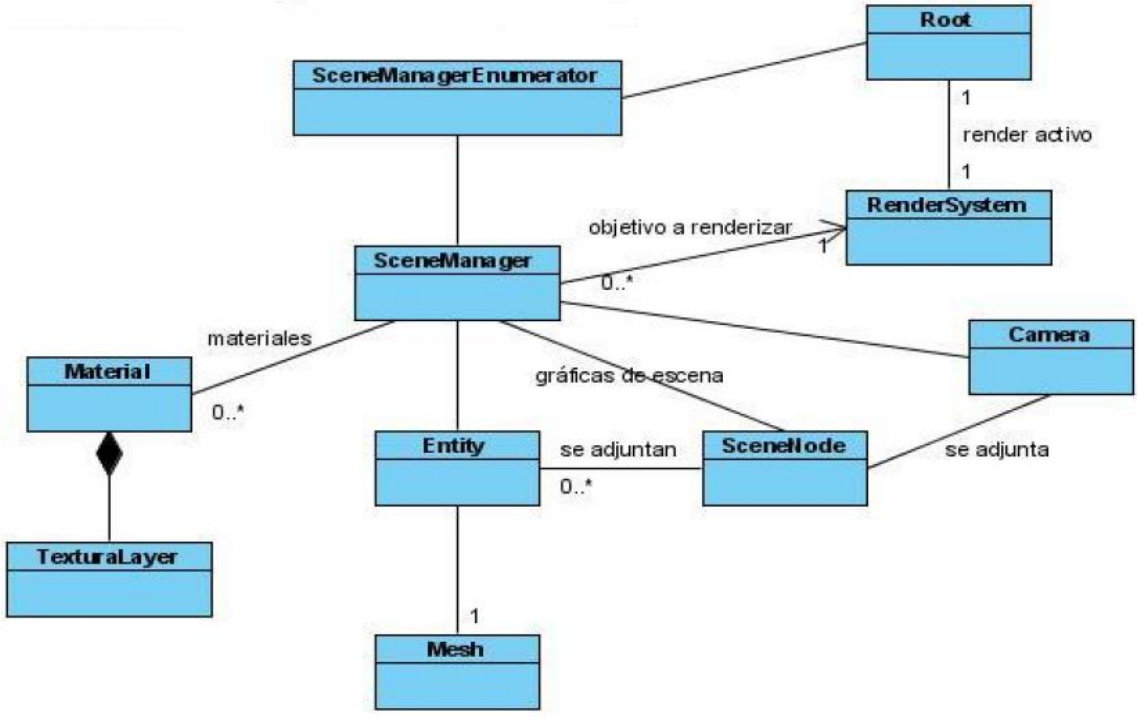

Figura 2.5 Diagrama de Clases de Ogre [20].

La instancia de la clase Root, es decir, el objeto “Root” es el punto de entrada al sistema de Ogre. Este debe siempre ser el primer objeto que se crea y el último que se destruye. Es el que permite configurar el sistema, selecciona (o da la posibilidad de seleccionar) el sistema para el renderizado de los objetos que puedan haber en una escena, es decir, selecciona las librerías de OpenGL o Direct3D dependiendo de que estas existan y/o de las preferencias del usuario. Inicializa también al SceneManager y posee un método llamado “startRendering()” que es el encargado de iniciar el renderizado de los objetos y mantener el renderizado en cada frame mediante un *loop* hasta que se le dé la orden de finalizar, esto es terminando la aplicación. Un *frame* puede considerarse como una imagen independiente, una sucesión de *frames* dan la sensación de movimiento o animación, la fluidez del movimiento o la animación dependerá de los *frames* que se alcancen a desarrollar en un segundo. El cine tradicional genera 24 *frames* por segundo para dar la sensación al ojo humano de movimiento fluido. Los *frames* por segundo que desarrolle una aplicación por computadora dependerán de los procesos y/o cálculos que se tengan que realizar en cada *frame* (en cada iteración dentro del *loop*) además de las capacidades de hardware (procesador, aceleradora de video) [20].

La clase RenderSystem es la que define las interfaces entre Ogre y las API 3D subyacentes (OpenGL, Direct3D). Una vez inicializado el sistema por el objeto “*Root*”, éste selecciona el API 3D y le indica a *RenderSystem* cuál es la API seleccionada para realizar el renderizado [20].

SceneManager es considerada la segunda clase más importante después de la clase *Root*. Es la clase encargada de organizar todos los elementos que están en una escena para ser renderizados. Una escena puede ser definida como la suma de todos los objetos que serán desplegados por pantalla.

El objeto *SceneManager* crea las cámaras, luces, objetos (entidades) de una escena y mantiene la pista de todos estos para poder acceder a ellos cuando se requiera para dar la posibilidad de manipularos según convenga. Cuando llega el momento de renderizar una escena *SceneManager* envía a *RenderSystem* todos los objetos que se deben mostrar. Existen diferentes tipos de “*SceneManager*”, algunos manejan mejor escenas en espacios cerrados, dentro de habitaciones o pasillos como por ejemplo juegos de primera persona tipo Doom. Otros están optimizados para escenas al aire libre. Pueden haber activos más de un “*scene manager*” al mismo tiempo para una misma aplicación. La clase ***SceneManagerEnumerator*** es la que tiene conocimiento de todos los “*SceneManager*” disponibles y activos [20].

La clase ***Mesh*** representa un modelo discreto, un set de geometrías autónomo que generalmente es más pequeño que el mundo que integra. Un objeto *mesh* se utiliza para representar objetos móviles dentro de la escena, generalmente son creados en herramientas de modelado 3D y luego son exportados a un archivo .mesh que utiliza Ogre para recrear el objeto y desplegarlo por pantalla. También

se pueden crear directamente en Ogre de forma manual realizando llamadas a métodos.

Los objetos de la clase ***Entity*** son instancias de un objeto móvil en la escena, estos pueden ser por ejemplo una persona, un perro, entre otros. Una entidad puede ser cualquier cosa y se basa en un set de geometrías, es decir, en los objetos *Mesh*. Una entidad entonces puede ser un vehículo modelado en una herramienta 3D, que es exportada a un archivo .mesh que es a su vez considerado por Ogre como un objeto *Mesh* [20].

La clase ***SceneNode***. Todas las entidades y opcionalmente cámaras y luces, son enlazadas o adjuntadas a un nodo, la forma de trabajar con una entidad u objeto de la escena es a través del nodo al que el objeto está enlazado. Las trasformaciones tales como rotación, traslación, orientación, movimiento, no se aplican directamente a la entidad sino que al nodo que posee, es el nodo el que se rota o mueve a una determinada posición dentro del espacio tridimensional. Los nodos poseen una estructura jerarquizada que es manejada por la clase *SceneManager*, quien los crea y los destruye, en donde un nodo puede tener un nodo padre (solo uno) y muchos nodos hijos. Al inicializar el sistema a través de Root y crear un objeto *SceneManager* se crea automáticamente con este último un nodo llamado "*nodo Root*" que es el nodo principal del cual se crean (ramifican) el resto de los nodos a los cuales serán enlazados los diferentes objetos que tenga una escena. La Figura 2.6 permite aclarar esta jerarquía [20].

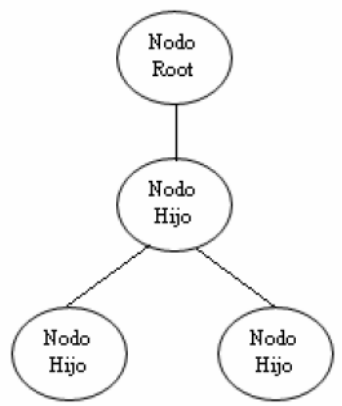

Figura 2.6 Jerarquía [20].

Una entidad solo puede ser enlazada a un solo nodo pero un nodo puede tener enlazada a él más de una entidad. El contenido de la escena que se renderizará en definitiva es el contenido de los nodos (o *SceneNodes)* que en su conjunto conforma la estructura de la escena gráfica o "gráficas de escena" que es enviada a la clase *SceneManager* para que ésta a su vez la envíe a *RenderSystem* para ser finalmente desplegado por pantalla. Así como un objeto puede enlazarse a un nodo también puede desligarse de él, si así fuera este objeto no será renderizado [20].

La clase ***Camera*** es la que, como su nombre lo describe, define los atributos y propiedades de las cámaras que serán las encargadas de entregar el punto de vista desde el cual se observará la escena renderizada. Las cámaras pueden ser rotadas, movidas directamente o a través de un nodo al que se puede enlazar [20].

La clase ***Material*** controla todo lo relacionado con el aspecto de los objetos, sin considerar su forma. Un objeto de la clase *Material* controla cómo los objetos en la escena son renderizados desde el punto de vista de la apariencia, especifica las propiedades básicas de la superficie de los objetos como la reflexión de los colores, el brillo, las capas de texturas, los efectos que son aplicados. Los materiales pueden ser aplicados programáticamente llamando al método *createMaterial* de la clase *SceneManager* o pueden ser cargados por la aplicación en tiempo de ejecución a través de un *script* que contiene toda la información necesaria del material, este *script* es un archivo en lenguaje intuitivo con extensión .material. Generalmente cuando se modela un objeto 3D en una herramienta modeladora se le definen también los materiales (color, texturas). Al momento de exportar el modelo 3D a un formato que Ogre puede interpretar se crean dos archivos uno con extensión .mesh que contiene la geometría del objeto 3D y otro con extensión .material que contiene la información de los materiales del objeto y las características de cómo serán aplicados. De esta manera cuando en Ogre se crea una “entidad” basada en un *mesh* se cargan automáticamente junto con ella los materiales, que son leídos desde el archivo .material [20].

Ogre provee un SDK (Kit de Desarrollo de Software), para desarrollar aplicaciones basadas en su motor gráfico que se puede descargar libremente desde su página web [21].

Ogre usa los ejes X y Z en el plano horizontal, y el eje Y en el eje vertical. Ahora en el monitor el eje X correría del costado izquierdo al derecho del monitor, el costado derecho seria la dirección positiva de X. El eje Y correría de abajo hacia arriba del monitor, la parte de arriba seria la dirección positiva de Y. El eje Z correría de la parte de adentro hacia afuera de la pantalla, la parte de afuera seria la dirección positiva de Z (Figura 2.7).

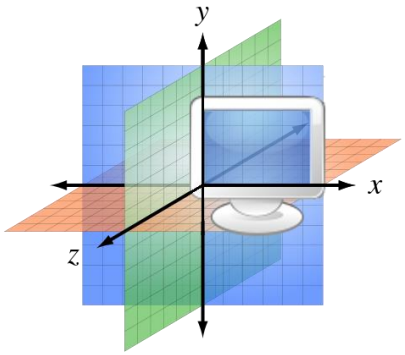

Figura 2.7 Ejes X, Y y Z.

2.2 Bibliotecas Hápticas

A continuación se presenta un resumen de las diferentes bibliotecas hápticas estudiadas para realizar la comunicación entre el motor de simulación 3D y Novint Falcon.

2.2.1 Novint HDAL

El objetivo principal de HDAL es proporcionar una interfaz uniforme para todos los tipos de dispositivos compatibles. El programador de la aplicación se libera de la responsabilidad de saber cómo se inicia cada dispositivo, cómo sus datos son recuperados y entregados, y por qué medios los cálculos de fuerza son inicializados.

El SDK contiene toda la documentación y los archivos de software necesarios para desarrollar aplicaciones con el dispositivo háptico Novint Falcon™ desde la capa de abstracción HDAL (*Haptic Device Abstraction Layer*) con el lenguaje de programación C y C++, ya que la última versión contiene completa compatibilidad con C. Para su uso, se asume que los controladores del Novint Falcon y del puerto USB ya han sido instalados, y ya se han incluido los archivos DLLs, NOVINT*.BIN, y HDAL en los proyectos de aplicación. Los ejemplos presentados en el SDK se han desarrollado para trabajar en los entornos de programación grafica DirectX[4] y OpenGL[5].

HDAL proporciona la posibilidad de generar interfaces de programación para el Novint Falcon, incluyendo tareas de inicialización, lectura de estado (posición, velocidad, botones, etc.), comandos de cálculo y aplicación de fuerzas a través de una función tipo *callback* ejecutada a una tasa de 1 KHz. para alcanzar un alto grado de fidelidad háptica. Los niveles de abstracción de HDAL están dispuestos de tal forma que permitan una sincronización correcta entre la aplicación gráfica y la realimentación de fuerzas tal como se indica en la Figura 2.8. La comunicación entre la capa de simulación háptica y las funciones de HDAL se realiza a través de una función *callback* invocada desde el interior de HDAL a mil veces por segundo ("*servo-tick*"). Dentro de esta función el usuario lee la posición del efector final del Novint Falcon, calcula los niveles de fuerza y los envía para ser aplicados al dispositivo.

Aunque por ahora solo se trabaja con el NOVINT FALCON, la compañía Novint Inc. planea construir otras versiones, por lo que HDAL ya incluye la gestión no solo de uno sino de toda una familia de controladores de dispositivos hápticos, cada uno de los cuales tendrá su propio SDK manejado a partir de una capa de control

[4] DirectX es una colección de API creadas y recreadas para facilitar las complejas tareas relacionadas con multimedia, especialmente programación de juegos y vídeo en la plataforma Microsoft Windows.
[5] OpenGL (Open Graphics Library) es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D.

de HDAL. La comunicación física con el dispositivo háptico (USB para el Novint Falcon) será responsabilidad del propio SDK que haya sido activado en la capa anterior [22].

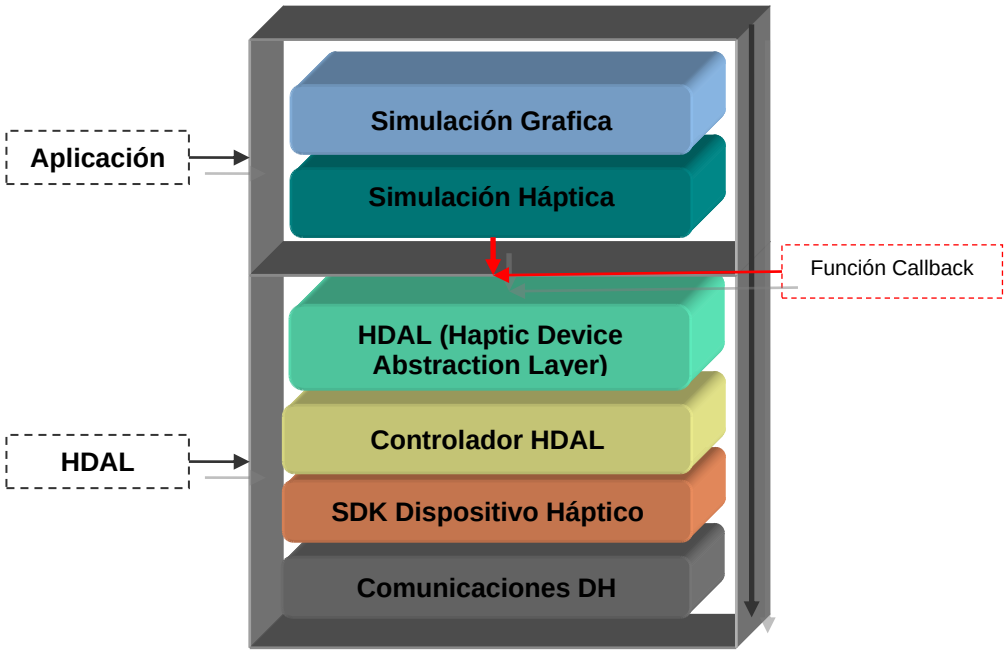

Figura 2.8. Novint HDAL SDK.

2.2.2 Libnifalcon

Libnifalcon es una biblioteca de desarrollo de software para la interfaz háptica Novint Falcon, y es una multiplataforma de fuente abierta alternativa a Novint HDAL SDK. Ésta provee comunicaciones y el modelo cinemático para el controlador de la interfaz. Libnifalcon proporciona una funcionalidad básica para conectar la interfaz háptica [23].

La principal meta del diseño de esta biblioteca es hacer un driver que sea tan flexible como el mismo hardware. La interfaz Novint Falcon provee características expandibles como carga del *firmware*[6] y cambio del agarre. Libnifalcon se diseñó con la misma idea [23].

Se puede acceder a Novint Falcon por medio de cuatro comportamientos:

- Comunicaciones - cómo se puede hablar con el hardware.

[6] Firmware es un programa que es grabado en una memoria ROM y establece la lógica de más bajo nivel que controla los circuitos electrónicos de un dispositivo. Se considera parte del hardware por estar integrado en la electrónica del dispositivo, pero también es software, pues proporciona la lógica y está programado por algún tipo de lenguaje de programación. El firmware recibe órdenes externas y responde operando el dispositivo.

- Firmware - cómo funciona el integrado de hardware y se comunica.
- Cinemática - la capacidad para obtener la posición del efector final y aplicarle fuerzas.
- Agarre - acceso a todas las características del agarre fabricado.

Libnifalcon posee un conjunto estable de clases que implementa los cuatro comportamientos. Cada uno de estos comportamientos puede ser cambiado cuando sea necesario a través de la clase *FalconDevice*, lo que indica que esta biblioteca puede ser utilizada para acceder a la interfaz háptica Novint Falcon así como para la investigación sobre el nuevo firmware, hardware de agarre y la cinemática del dispositivo [23].

Entre algunas posibles aplicaciones de Libnifalcon se encuentran:

- Educación
- Ingeniería mecánica (dinámica).
- Ingeniería en informática (controladores de hardware, firmware / dsp desarrollo)
- Ciencias de la computación (controladores de hardware, software de organización)
- Interacción hombre - máquina (tacto, el desarrollo de agarre).

Es importante notar que si se es nuevo en el campo de la háptica, libnifalcon es simplemente un controlador para un determinado hardware háptico. No es un motor grafico para la háptica. Esto significa que, si bien se tienen ejemplos de aplicación de colisión con un poco de geometría simple, no es el verdadero significado para el uso de la investigación háptica a menos que se esté buscando hacer cinemática muy específica [23].

2.2.3 H3DAPI

H3DAPI es una plataforma de fuente abierta para desarrollo de software que utiliza los estándares abiertos OpenGL y X3D[7] en un escenario gráfico unificado para manejar gráficos y el tacto.

[7] X3D es un lenguaje informático para gráficos vectoriales definido por una norma ISO, que puede emplear tanto una sintaxis similar a la de XML como una del tipo de VRML (Virtual Reality Modelling Language). X3D amplía VRML con extensiones de diseño y la posibilidad de emplear XML para modelar escenas completas en tiempo real.

A diferencia de la mayoría de los interfaces de escenas gráficas, H3DAPI está diseñada principalmente para apoyar un proceso especial de desarrollo rápido. Mediante la combinación de X3D, C + + y el lenguaje de script Python, H3DAPI ofrece tres modos de programación de aplicaciones que ofrecen lo mejor de ambos mundos - la velocidad de ejecución donde el rendimiento es crítico, y la velocidad de desarrollo donde el rendimiento es menos crítico [24].

H3DAPI está escrito en C + +, está diseñado para ser extensible, asegurando que los desarrolladores tengan la libertad y los medios para personalizar y añadir cualquier realimentación táctil necesaria o características gráficas en H3DAPI para sus aplicaciones. Se ha desarrollado una amplia gama de aplicaciones con H3DAPI en diversas áreas como medicina, industria y visualización [24].

H3DAPI es de licencia dual, de código abierto y de plataforma comercial, tiene una escena gráfica API disponible para su descarga [24]. Pueden ser creados mundos virtuales animados a partir de la utilización de las aplicaciones que contiene la sintaxis X3D. Se pueden crear desde el mundo de Python animaciones y comportamientos más avanzados. Estos mundos contienen tanto escenas gráficas como realimentación del tacto.

A través de la utilización del concepto gráfico de escena en X3D, los mundos virtuales pueden ser fácilmente definidos. Debido al concepto de escenario gráfico, siempre es fácil obtener una visión general de cómo el mundo virtual se define. Los usuarios con poca experiencia en programación de bajo nivel pueden configurar las escenas simples que podrían ser utilizadas para la investigación experimental [24].

H3DAPI viene con una serie de ejemplos sencillos que muestran las características de la biblioteca. H3DAPI utiliza la sintaxis X3D y el concepto de nodos para construir el mundo virtual. Un nodo proporciona una característica determinada en la escena. Hay nodos para la renderización de la geometría en el mundo virtual, para la creación de gráficos y propiedades hápticas y los nodos que pueden ser utilizados para la animación [24].

H3DAPI es independiente del dispositivo háptico (a través de HAPI) y soporta múltiples dispositivos comerciales disponibles en la actualidad entre ellos el Novint Falcon. Mediante el uso de la interfaz de AnyDevice se puede utilizar cualquiera de los dispositivos compatibles conectados [24].

Esta biblioteca contiene un par de maneras diferentes para generar fuerzas de mallas de superficie 3D. Para generar la fuerza de gráficos en 3D un procesador háptico debe ser elegido. A través de X3D puede sentirse diferentes geometrías dependiendo de qué nodo de superficie se esté utilizando y cómo las propiedades del nodo son ajustadas [24].

2.2.4 OgreHaptics v 2.0

OgreHaptics es una biblioteca para desarrollo de software la cual está escrita en C++ para la integración de Ogre 3D con dispositivos hápticos disponibles en el mercado, lo que permite al usuario manipular los entornos virtuales en 3D usando retroalimentación de fuerza.

Está diseñado para hacer más fácil y más intuitivo a los desarrolladores el producir demos y juegos que utilizan gráficos 3D y dispositivos hápticos.

El renderizado háptico, o el renderizado de fuerzas, puede ser realizado de diferentes formas. El renderizado de una superficie en la cual una fuerza generada por la colisión entre el efector final y un objeto virtual es mostrada al usuario, es una de las formas más interesantes de interacción háptica. Otra forma es el renderizado de los efectos de fuerza del ambiente como los campos de fuerza o la viscosidad del espacio virtual a través del cual el usuario mueve el efector final. La versión actual de OgreHaptics solo implementa el renderizado de efectos de fuerza y el renderizado de superficies está planeado para la próxima versión [25].

Esta biblioteca implementa los medios para proveer hilos seguros para la sincronización de datos entre el cliente y el dispositivo háptico. Esto asegura que los datos usados por el dispositivo tanto como por el cliente pueden ser usados seguramente por el usuario.

La Figura 2.9 presenta un diagrama de clases donde se pueden apreciar las clases más importantes de la biblioteca OgreHaptics:

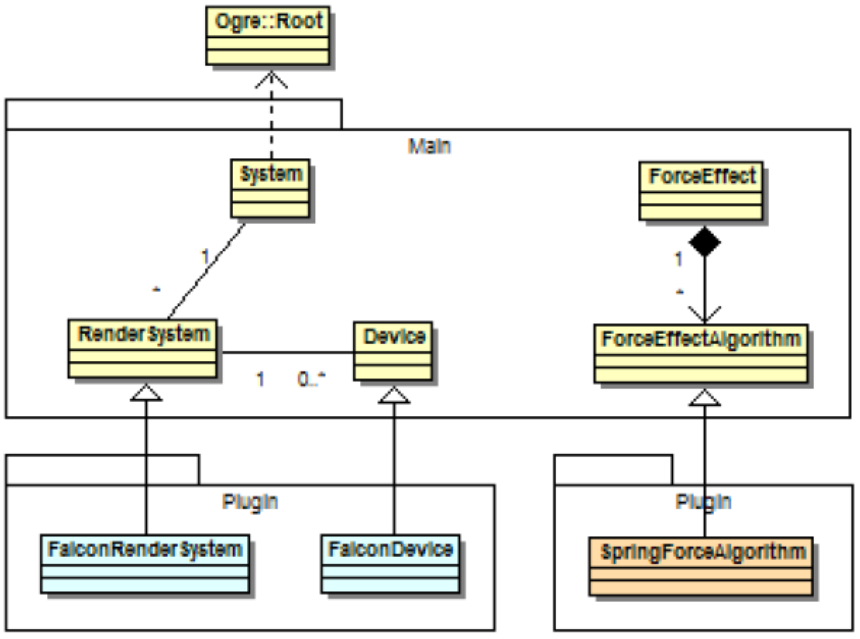

Figura 2.9 Diagrama de clases OgreHaptics.

A continuación se muestra una corta descripción de los objetos principales que permiten la realización del renderizado háptico:

El objeto *System* es el punto de entrada principal al sistema de OgreHaptics. Este objeto debe ser el primero en ser creado y el ultimo en ser destruido. Se basa en la clase *Ogre::Root* de la cual se crea una instancia. Atraves de los plugins del objeto *System* se pueden añadir otros dispositivos hápticos para ser usados en las aplicaciones [25].

La clase *RenderSystem* es una clase de base abstracta, describe la interfaz para una API háptica. Está actualmente implementada para la interfaz háptica Phantom y tambien puede ser usada con Novint Falcon. Una aplicación típica no se comunica directamente con este objeto a menos que se necesite acceder a métodos más complejos utilizados por el hilo háptico [25].

A través del objeto *Device* se pueden controlar la entrada y salida de un dispositivo háptico, como por ejemplo los ajustes que se le hacen al espacio háptico para ser mapeados al espacio del dispositivo, la consulta del estado actual del dispositivo y el registro de eventos generados por la interfaz háptica [25].

El objeto *ForceEffect* provee los medios para crear fuerzas ambientales como viscosidad, resortes y vibración. Estas fuerzas son ambientales porque no son relacionadas con ninguna entidad o forma en el ambiente virtual. Un efecto de fuerza consiste en uno o mas de las subclases de los algoritmos de fuerza las cuales implementan algoritmos actuales para el renderizado de fuerzas [25].

2.3 Posicionamiento de un objeto usando Novint Falcon

En este proyecto se utilizó Microsoft Visual Studio 2005 como plataforma, con lenguaje de programación visual C++. Para el ambiente tridimensional virtual el motor de renderizado gráfico de código abierto es Ogre 3D v1.6.4.

De las bibliotecas mencionadas, después de estudiarlas detenidamente, se escogió OgreHaptics v2.0, ya que brinda la oportunidad de usar directamente OGRE 3D, el software utilizado para crear la interfaz gráfica en la cual está construido LapBot.

El kit para desarrollo de software OgreHaptics v2.0 provee dos demos a partir de los cuales el programador puede iniciar su proyecto. Para este caso se hizo uso de demo Effects y demo Weapons.

Gracias a la exploración de la interfaz de programación de aplicaciones (API) presentada por OgreHaptics se encontró la función *getPosition* (), que retorna la posición del dispositivo en coordenadas del espacio de trabajo del mismo.

La primera prueba consistió en mover una esfera por medio de la interfaz háptica en Ogre 3D (Ver Figura 2.10).

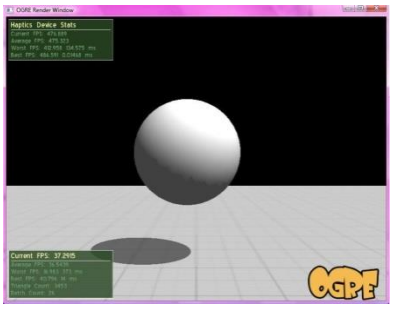

Figura 2.10 Primera prueba.

A continuación se presenta un diagrama de flujo (Figura 2.11) que muestra como se desarrolló el programa para posicionar la esfera por medio de la interfaz Háptica.

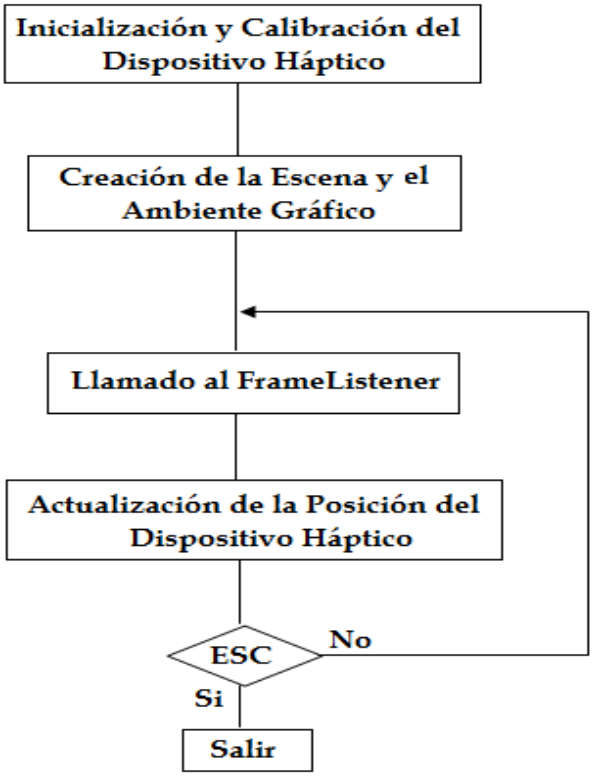

Figura 2.11 Diagrama de Bloques Primera Prueba.

2.3.1 Inicialización y Calibración del dispositivo háptico

Al incluir los siguientes archivos de cabecera se asegura que el dispositivo se inicie y se calibre correctamente:

```
#include "OgreHapticsDemoApplication.h"
```

Dentro de este archivo se encuentra la siguiente función que permite inicializar la interfaz:

```
mDevice = mSystem->initialise(true);
```

El siguiente archivo incluye las funciones necesarias para la calibración del dispositivo:

```
#include "OgreHapticsDemoFrameListener.h"

virtual void calibrationStateChanged(const OgreHaptics::DeviceEvent& evt)
```

Este archivo de cabecera incluye todos los archivos necesarios para construir una aplicación que utilice la biblioteca OgreHaptics

```
#include "OgreHaptics.h"
```

2.3.2 Creación de la Escena y del Ambiente Gráfico

Se creó en el motor de renderizado Ogre 3D una escena simple que consta de un piso y una esfera. Con las siguientes lineas de codigo se muestra la creación de la escena:

```
// Creación del piso
    Entity* ent = mSceneMgr->createEntity("Floor", "FloorPlane");
    ent->setMaterialName("OgreHaptics/Examples/FloorPlane");
    ent->setCastShadows(false);

// Creación de la esfera
    SceneNode* node = mSceneMgr->getRootSceneNode()->createChildSceneNode();
    gCursor = mSceneMgr->createEntity("Cursor", "sphere.mesh");
    gCursor->setMaterialName("Plain/White");
    node->attachObject(gCursor);
    node->setScale(0.5, 0.5, 0.5);
```

Primero se crea una entidad (createEntity) que contiene la imagen del cuerpo (sphere.mesh), luego se asigna el material que se desea para dicho cuerpo (setMaterialName), y se crea un nodo de movimiento que tiene una relación padre-hijo con el nodo de la escena (createChildSceneNode). Se enlaza la entidad con el nodo para que se muevan juntos en la simulación (attachObject). Se pueden colocar

sombras al objeto (setCastShadows) adhiriéndolas a la entidad y también se puede escalar el objeto (setScale) por medio del nodo.

2.3.3 Llamado al FrameListener

Mediante la siguiente función se realiza un llamado a la clase EffectsListener, para que se actualice en cada frame.

```
void createFrameListener(void)
        {
                EffectsListener* effectsListener = new EffectsListener(mWindow,  mCamera,
                mSystem, mDevice);
                mFrameListener = effectsListener;
                mFrameListener->showDebugOverlay(true);
                mRoot->addFrameListener(mFrameListener);
        }
```

2.3.4 Actualizacion de la Posición del Dispositivo Háptico

Al llamar al FrameListener se accede a la clase EffectsListener dentro de la cual se actualizan todos los componentes del ambiente, entre ellos la posición del dispositivo háptico, en este caso la posicion de la esfera mediante la siguiente linea de codigo:

```
gCursor->getParentSceneNode()->setPosition(mDevice->getWorldPosition());
```

En la cual se adquiere la posicion de la interfaz háptica y después se le pasa esta posición a la esfera.

Para finalizar el programa se presiona la tecla ESC.

2.3.5 Resultados

A continuacion se muestran imagenes (Figura 2.12) tomadas en diferentes instantes de tiempo que permiten ver a la esfera posicionada, a traves de Novint Falcon, en distintos puntos de la escena.

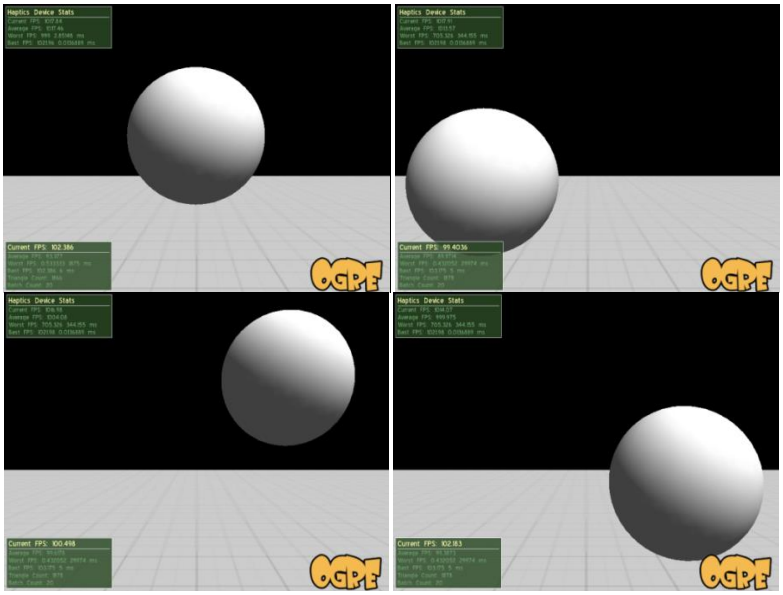

Figura 2.12 Posicionamiento de un objeto.

La Figura 2.13 muestra el movimiento de la esfera en el plano Z de la interfaz haptica:

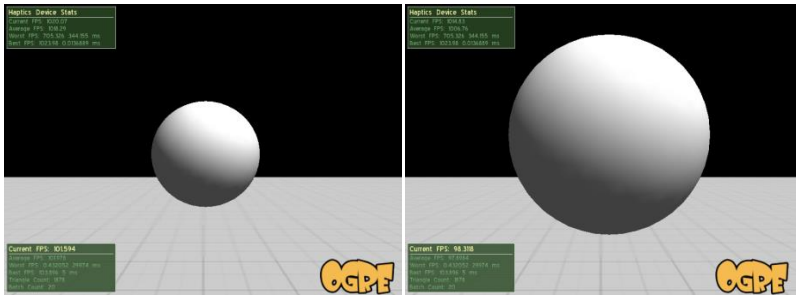

Figura 2.13 Movimiento en el plano Z del dispositivo háptico.

2.4 Posicionamiento de LapBot

Después de realizar la primera prueba, para efectuar la comunicación entre la interfaz háptica y Ogre3D, se incluyó en el ambiente la sala de operaciones, la camilla, la cámara, las luces, el abdomen simulado del paciente y el robot para cirugía laparoscópica LapBot.

A continuación se muestra el diagrama de flujo (Figura 2.14) que explica los pasos seguidos para realizar el posicionamiento de LapBot por medio de Novint Falcon:

Figura 2.14 Diagrama de Bloques del Programa de Posicionamiento de LapBot.

2.4.1 Inclusión de la escena quirúrgica en el ambiente 3D

Primero se construye en Ogre 3D la sala de operaciones, en la cual se ubicará el robot, la camilla y el abdomen simulado, con las siguientes líneas de código:

Para la habitación:

```
// Habitacion
MaterialPtr mpiso = MaterialManager::getSingleton().create("Mpiso",
ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);
TextureUnitState* tpiso = mpiso->getTechnique(0)->getPass(0)->createTextureUnitState("wood_15.jpg");
tpiso->setTextureScale(0.05,1);

Entity *piso = mSceneMgr->createEntity("piso",mSceneMgr->PT_CUBE);
piso ->setMaterialName("Mpiso");
SceneNode *npiso = nodoRoom->createChildSceneNode(Vector3(1,-3.35,0));
npiso->attachObject(piso);
npiso->scale(0.2, 0.0015, 0.2);

// Generando las paredes
MaterialPtr mpared = MaterialManager::getSingleton().create("mpared",
ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);
TextureUnitState* tpared = mpared->getTechnique(0)->getPass(0)->createTextureUnitState("marble_7.jpg");
```

```
Entity *pared = mSceneMgr->createEntity("pared",mSceneMgr->PT_CUBE);
pared->setMaterialName("mpared");
SceneNode *npared = npiso->createChildSceneNode(Vector3(50,4000,0));
npared->attachObject(pared);
npared->roll(Degree(90));
npared->scale(0.6,1,1);

Entity *pared2 = pared->clone("pared2");
SceneNode *npared2 = npiso->createChildSceneNode(Vector3(-50,4000,0));
npared2->attachObject(pared2);
npared2->roll(Degree(90));
npared2->scale(0.6,1,1);

Entity *pared3 = pared->clone("pared3");
SceneNode *npared3 = npiso->createChildSceneNode(Vector3(0,4000,-50));
npared3->attachObject(pared3);
npared3->pitch(Degree(90));
npared3->scale(1,1,0.6);

Entity *pared4 = pared->clone("pared4");
SceneNode *npared4 = npiso->createChildSceneNode(Vector3(0,4000,50));
npared4->attachObject(pared4);
npared4->pitch(Degree(90));
npared4->scale(1,1,0.6);

// Generando el techo
Entity *techo = pared->clone("techo");
SceneNode *ntecho = npiso->createChildSceneNode(Vector3(0,8000,0));
ntecho->attachObject(techo);
```

Como se observa se realiza primero el piso, luego las paredes y por último el techo; a cada entidad se le puede asignar un material diferente para formar el diseño de habitación que se desee, en el caso de este proyecto se seleccionó piso de madera y lo demás color mármol. La creación de materiales se realiza con estos comandos [4]:

```
MaterialPtr mpared =   MaterialManager::getSingleton().create("mpared",
ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);
TextureUnitState* tpared = mpared->getTechnique(0)->getPass(0)->
createTextureUnitState("marble_7.jpg");
```

También se colocaron tres luces, de tal forma que se puedan observar los objetos, de lo contrario todo estaría negro. El comando para crear una luz es:

```
Light  *l1= mSceneMgr->createLight("Luz1");
```

Despues se incluye la camilla grande de dimensiones estándar con 0.6 m de ancho, 2 m de largo, altura respecto al piso de 0.6 m, y un cuerpo estilo caja para representar el abdomen insuflado del paciente con las respectivas incisiones.

El código en C++ se muestra a continuación:

```
//Mesa
Entity *Mesa  = mSceneMgr->createEntity("Mesa", "mesa.mesh");
Mesa ->setMaterialName("MatMesa");
SceneNode *nodoMesa = nodoRoom->
createChildSceneNode( Vector3(-1.325,-2.65,0));
nodoMesa->attachObject(Mesa);
nodoMesa->pitch( Degree(-90));

Entity *Abdomen = mSceneMgr->createEntity("Abdomen", "Mesh.mesh");
Abdomen ->setMaterialName("MatAbdomen");
```

```
SceneNode *nodoAbdomen=mSceneMgr->getRootSceneNode()->createChildSceneNode(Vector3(10.475,-2.65,8.8));
nodoAbdomen->attachObject(Abdomen);
```

Los robots se adicionan al final como nodos hijos de la mesa y se les adiciona los ejes coordenados.

Para la construcción de los robot se utiliza el siguiente codigo, por ejemplo para la articulacion r1:

```
Entity *r1 = mSceneMgr->createEntity("r1", "r1.mesh");
r1 ->setMaterialName("MatExterno");
nodor1 = r1s->createChildSceneNode(Vector3( 0, 0, 0));
nodor1 ->attachObject(r1);
```

Para el brazo del robot:

```
Entity *Brazo = mSceneMgr->createEntity("Brazo", "Brazo.mesh");
Brazo ->setMaterialName("MatExterno");
nodoBrazo = nodor1->createChildSceneNode(Vector3( 0,0,17.12));
nodoBrazo ->attachObject(Brazo);
```

Para el antebrazo:

```
Entity *Antebrazo = mSceneMgr->createEntity("Antebrazo", "Antebrazo.mesh");
Antebrazo ->setMaterialName("MatExterno");
nodoAntebrazo = nodoBrazo->createChildSceneNode(Vector3(9.84,0,-0.3));
nodoAntebrazo->attachObject(Antebrazo);
```

Primero se crea una entidad (createEntity) que contiene la imagen del cuerpo (r1.mesh), luego se asigna el material que se desea para dicha imagen (setMaterialName), y se crea un nodo de movimiento que tiene una relación padre-hijo con el nodo del cuerpo anterior (createChildSceneNode). Se tiene la posibilidad de ubicar el nuevo nodo en una posición específica deseada (Vector3) respecto al nodo padre. Por último se enlaza la entidad con el nodo para que se muevan juntos en la simulación (attachObject).

Es necesario enlazar los cuerpos de la forma padre-hijo para que al mover una articulación padre las hijas también se muevan, como sucede en la realidad, de esta forma se logra que el robot sea una pieza completa, conjunta y articulada.

Este procedimiento se realiza con cada cuerpo hasta armar completamente el robot. En algunos cuerpos también es necesario rotar un nodo usando los comandos pitch, yaw, roll, dependiendo del eje con respecto al cual se hace la rotación [4].

La Figura 2.15 permite ver la inclusión de la escena quirúrgica en el ambiente de Ogre, se puede apreciar el cuarto, la mesa, el abdomen y el robot LapBot.

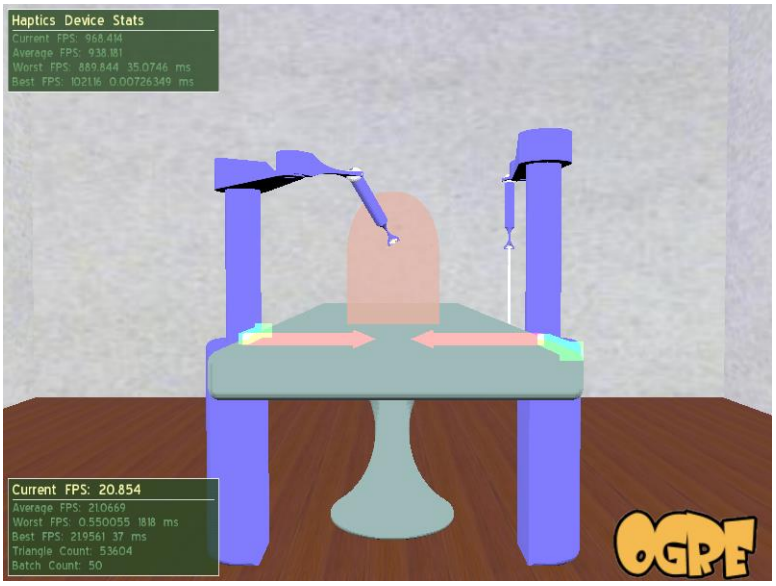

Figura 2.15 Ambiente 3D.

En la Figura 2.16 se puede apreciar, en diferentes vistas, la escena quirúrgica diseñada para esta aplicación:

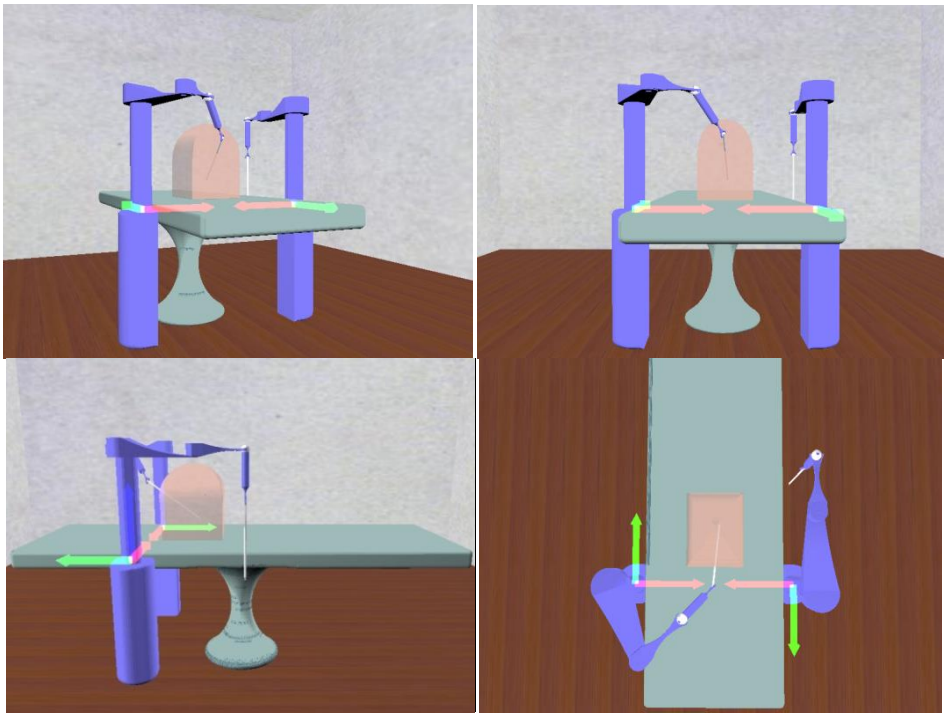

Figura 2.16 Vistas de la escena quirúrgica.

2.4.2 Inicialización y Calibración del Dispositivo Háptico

Como se mencionó en la sección 3.2.1 con la inclusión de los siguientes archivos de cabecera se puede realizar la inicialización y calibración de la interfaz háptica:

```
#include "OgreHapticsDemoApplication.h"

#include "OgreHapticsDemoFrameListener.h"

#include "OgreHaptics.h"
```

Al correr el ejecutable, la ventana de Ogre solicita la calibración de la interfaz háptica así como se muestra en la Figura 2.17:

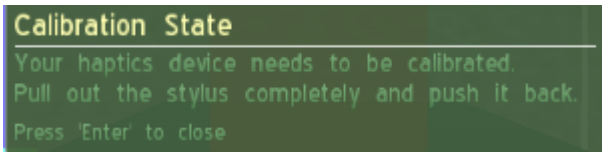

Figura 2.17 Calibración de la interfaz háptica

Si el dispositivo fue calibrado correctamente aparece un mensaje como se muestra en la Figura 2.18:

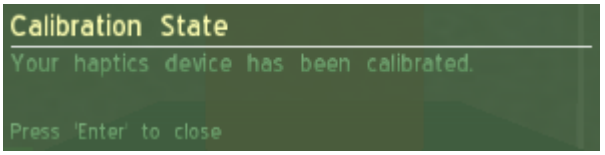

Figura 2.18 Dispositivo Calibrado

2.4.3 Llamado al FrameListener

El llamado al *FrameListener* se hace de igual forma como se muestra en el ítem 2.3.3.

En Ogre se puede registrar una clase para recibir una notificación antes y después que un *frame* es renderizado a la pantalla. Esta clase en Ogre es conocida como *FrameListener*.

La interfaz del *FrameListener* declara tres funciones las cuales pueden ser usadas para recibir eventos del *frame*.

```
virtual bool frameStarted(const FrameEvent& evt);
virtual bool frameRenderingQueued(const FrameEvent& evt);
virtual bool frameEnded(const FrameEvent& evt);
```

En este caso se hace uso de *frameRenderingQueued* para la actualización de los dispositivos correspondientes y se utiliza también *frameEnded*.

2.4.4 Lectura del Teclado y Ratón. Generación de la Acción Deseada

Para la lectura del teclado y del ratón se hace uso de la biblioteca OIS que permite el uso del teclado y el ratón en una aplicación. Se hace uso de esta biblioteca en el *FrameListener* para que constantemente se actualice la lectura de estos dispositivos.
Por medio de los siguientes comandos se realiza la lectura del teclado y el ratón:

```
mMouse->capture();
mKeyboard->capture();
```

Despues para la interacción del usuario con estos dispositivos se utilizan las siguientes funciones que son heredadas de la clase *ExampleFrameListener* de Ogre3D:

La Función *processUnbufferedMouseInput* permite el movimiento de la camara con el raton.

```
virtual bool processUnbufferedMouseInput(const FrameEvent& evt)
        {

// Rotation factors, may not be used if the second mouse button is pressed
                // 2nd mouse button - slide, otherwise rotate
                const OIS::MouseState &ms = mMouse->getMouseState();
                if( ms.buttonDown( OIS::MB_Right ) )
                {
                        mTranslateVector.x += ms.X.rel * 0.13;
                        mTranslateVector.y -= ms.Y.rel * 0.13;
                }
                else
                {
                        mRotX = Degree(-ms.X.rel * 0.13);
                        mRotY = Degree(-ms.Y.rel * 0.13);
                }

                return true;
        }
```

La funcion processUnbufferedKeyInput lee si alguna tecla ha sido presionada por el usuario y ejecuta una accion predeterminada, por ejemplo si el usuario presiona la tecla escape la funcion termina el programa. Con las teclas W,S,A y D se puede manejar la camara en la escena.

```
virtual bool processUnbufferedKeyInput(const FrameEvent& evt)
{

    if(mKeyboard->isKeyDown(OIS::KC_A))
    mTranslateVector.x = -mMoveScale;     // Move camera left

    if(mKeyboard->isKeyDown(OIS::KC_D))
     mTranslateVector.x = mMoveScale;     // Move camera RIGHT

  if(mKeyboard->isKeyDown(OIS::KC_UP) || mKeyboard->isKeyDown(OIS::KC_W)
       mTranslateVector.z = -mMoveScale;   // Move camera forward

  if(mKeyboard->isKeyDown(OIS::KC_DOWN)||mKeyboard->isKeyDown(OIS::KC_S))
       mTranslateVector.z = mMoveScale;     // Move camera backward

  if(mKeyboard->isKeyDown(OIS::KC_PGUP))
       mTranslateVector.y = mMoveScale;     // Move camera up
```

```
if(mKeyboard->isKeyDown(OIS::KC_PGDOWN))
    mTranslateVector.y = -mMoveScale;    // Move camera down

if(mKeyboard->isKeyDown(OIS::KC_RIGHT))
            mCamera->yaw(-mRotScale);

if(mKeyboard->isKeyDown(OIS::KC_LEFT))
            mCamera->yaw(mRotScale);

if(mKeyboard->isKeyDown(OIS::KC_ESCAPE)||mKeyboard->isKeyDown(OIS::KC_Q))
                return false;

            // Return true to continue rendering
            return true;
  }
```

2.4.5 Lectura de la posición de la interfaz háptica.

Para leer la posición de la interfaz háptica se hizo uso de la interfaz de programación de aplicaciones (API) presentada por OgreHaptics y se encontró la función *getPosition ()*, que retorna la posición del dispositivo en coordenadas del espacio de trabajo del mismo.

Con la siguiente línea de código se obtiene la posición del dispositivo háptico y se almacena en un vector:

```
Vector3 devicePos = mDevice->getPosition();
```

2.4.6 Implementación del MGI

El brazo robótico de LapBot tiene nueve articulaciones que deben moverse armónicamente de acuerdo al movimiento del efector final y respetando el punto de incisión del abdomen simulado del paciente. Este efecto se consigue gracias a la implementación del Modelo Geométrico Inverso (MGI). El MGI permite calcular los valores de las variables θ_j (variable articular de rotación [rad]) y r_j (variable articular de traslación [m]) asociadas a las articulaciones, dependiendo de la orientación y localización deseada del efector final del robot en el espacio cartesiano (x, y, z).

El MGI del robot fue modificado debido a que se hizo en Matlab y era necesario pasarlo a visual C++. Se cambiaron las funciones para los cálculos matemáticos como la potenciación y radicación las cuales son diferentes a las presentadas por Matlab.

Una vez leída la posición de la interfaz háptica mediante `mDevice->getPosition();`, se almacena en un vector `Vector3 devicePos`, luego se extraen las posiciones de x,y,z almacenadas en este vector para que puedan ser usadas en los calculos del MGI asi:

```
x1 = devicePos.x;
y1 = devicePos.y;
```

```
z1 = -devicePos.z;
```

Estos valores fueron escalizados debido a que el espacio de trabajo de la interfaz haptica es diferente al usado por el mundo virtual de Ogre 3D. La escalización se muestra a continuación:

```
x = 0.345+x1/400;
y = 0.1952+z1/357;
z = 0.2+y1/450;
```

Se puede observar que fue necesario cambiar los ejes Y y Z para que coincidieran con los del mundo virtual.

Con la escalización hecha se realizan los cálculos para obtener los valores de las variables θ_j y r_j asociadas a las articulaciones del robot para la generación del movimiento.

Por último fue necesario realizar cambios a la matriz de orientación del robot ya que el efector final estaba orientado hacia arriba en la simulacion en Ogre3D y se necesita que la orientación de la pinza sea hacia abajo.

2.4.7 Generación del Movimiento a partir del MGI

Con las siguientes líneas de código se genera el movimiento del robot mediante la interfaz háptica a partir de los cálculos realizados en el MGI.

```
// para r1
nodor1->setPosition(0,0,r1*40);

// para t2
nodoBrazo->setOrientation(Quaternion(Radian(t2),Vector3( 0,0,1)));

// para t3
nodoAntebrazo->setOrientation(Quaternion(Radian(t3),Vector3(0,0,1)));

//t4 es fija

// para t5
nodot5->setOrientation(Quaternion(Radian(t5),Vector3(0,0,1)));

// para t6
nodot6->setOrientation(Quaternion(Radian(t6),Vector3(0,0,1)));

// para t7
nodot7->setOrientation(Quaternion(Radian(t7),Vector3(0,0,1)));

// para t8
nodot8->setOrientation(Quaternion(Radian(t8),Vector3(0,0,1)));

// para t9
nodot9->setOrientation(Quaternion(Radian(t9),Vector3(0,0,1)));
```

Dado que *r1* es la unica articulacion de translación se utiliza para ésta la función *setPosition* para ubicarla en las coordenadas cartesianas (0,0,z) ya que solo se mueve alrededor del eje z.

Para las demás articulaciones se usa la función *setOrientation* la cual recibe un ángulo en radianes que se calcula en el MGI para cada articulación y el vector de posición (0, 0, 1) para que todas se muevan alrededor del eje z.

Despues de realizar la primera prueba se identificaron singularidades para las cuales el sistema no tenía solución. Fue necesario hacer un estudio que permitiera solucionar este problema. Se implementaron nuevas ecuaciones donde se consideraban soluciones que no se habian tenido en cuenta. Por ejemplo, cuando se calcula una raiz cuadrada se tienen dos soluciones posibles, una positiva y una negativa, inicialmente se habia considerado la solucion positiva, luego se incluyó una condición que permite activar la solución negativa para determinados casos, como se muestra a continuación:

```
if (z>tz)
   r1=-sqrt(((pow(R2,2))*N)/M) - K;
else
   r1 = sqrt(((pow(R2,2))*N)/M) - K;
```

Para ver las ecuaciones que se añadieron dirijase al anexo C donde se muestra el archivo Iteracion.h

2.4.8 Resultados

En la Figura 2.19 se muestra a LapBot siguiendo diferentes trayectorias y respetando siempre el paso por el trocar:

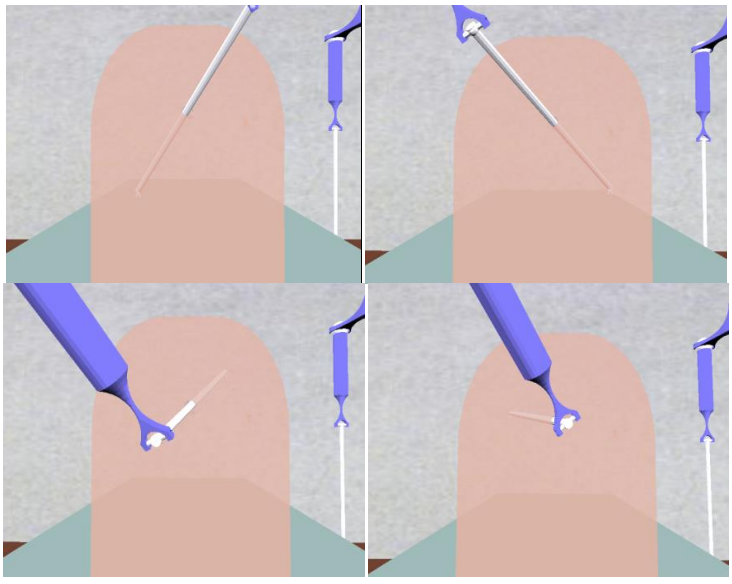

Figura 2.19 Paso por el trocar.

En la Figura 2.20 se puede observar como las articulaciones del brazo robótico se mueven para llevar al efector final a la posición indicada por el usuario a través de la interfaz háptica.

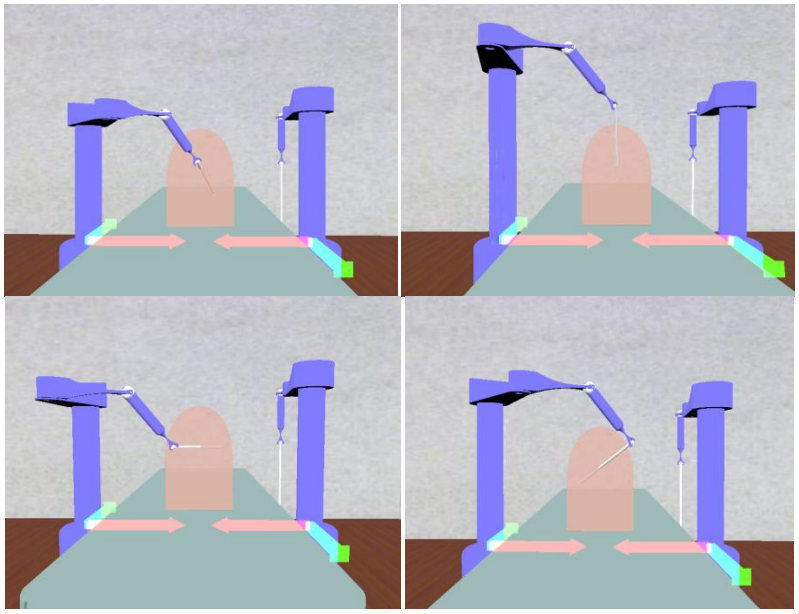

Figura 2.20 Movimiento de articulaciones.

3. Detección de Colisiones y Realimentación de Fuerza en un Ambiente Tridimensional Virtual

Después de lograr el posicionamiento del robot para cirugía laparoscópica LapBot en el ambiente tridimensional virtual con la interfaz háptica se procede a realizar la detección de colisiones entre el efector final del robot (pinza) y el abdomen simulado del paciente. Una vez detectadas las colisiones se realizará la realimentación de fuerza a la interfaz háptica para simular la restricción que constituye la pared abdominal al cirujano.

3.1 Algoritmos Para Detección de Colisiones

El objetivo principal de estos algoritmos es calcular las interacciones geométricas entre los objetos, sin importar el número ni la complejidad que los objetos puedan tener. Tradicionalmente han requerido una gran cantidad de pruebas de intersección geométrica, verificando si todos los polígonos que modelan toda la superficie de un objeto, interceptan la superficie de otro objeto, determinando de esta manera si dos objetos colisionan. La clave para la detección de colisiones en tiempo real, es el método que permita detectar más rápidamente cuando dos objetos no presentan colisión [2].

La mayoría de investigadores en el área proponen algoritmos que ya han dado resultados efectivos y reducen el número de llamados para verificar la intersección entre dos primitivas geométricas. Estas técnicas plantean un tipo de volúmenes envolventes organizados en una estructura jerárquica y con ello evitando una verificación de los pares de primitivas geométricas directamente [2].

Para la detección de colisiones en un ambiente virtual con realimentación de fuerzas, podrían usarse tres técnicas: subdivisión espacial jerárquica, subdivisión de objetos jerárquicos y cálculo de distancia incremental. La subdivisión espacial jerárquica es una técnica recursiva de particionamiento que divide el ambiente en

segmentos, con la ventaja de que si un objeto cambia su posición, solamente se debe chequear la colisión en el nuevo espacio donde se ha ubicado [2].

Las estructuras jerárquicas usadas en la detección de colisiones incluyen árboles de conos, esferas o cubos, entre otros, que llevan a cabo muy bien las pruebas de rechazo, cuando los objetos están suficientemente separados. Estas estructuras a su vez se pueden dividir en dos grupos: la jerarquía de la subdivisión de espacio y la jerarquía de la subdivisión del objeto [2].

La subdivisión de objetos consiste en encerrar los sólidos de entorno en un volumen jerárquico como esferas o paralelepipedos. La colisión se produce cuando se detecta que los volúmenes de contorno se intercepten entre sí [2]. Dependiendo del volumen seleccionado se pueden aplicar técnicas como: *Oriented Bounded Box* (OBB), *BoundingSphere* (BS) o Axis *Aligned Bounded Box* (AABB), tal como se muestra en la Figura 3.1. Algunas técnicas se explican a continuación:

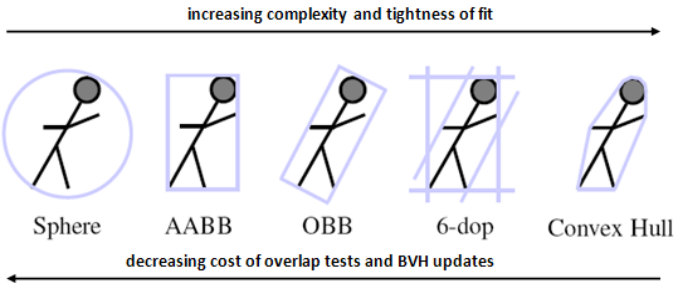

Figura 3.1: Clases de Volúmenes envolventes [2].

3.1.1 Técnica de *Bounding Sphere*.

A través de la técnica de BS o *Bounding Sphere*, los objetos colisionables se envuelven en esferas cuyo radio se determina por la máxima coordenada del objeto sólido. La detección de colisiones con esferas es más fácil de determinar, ya que solo basta comparar la distancia entre los centros de esferas, con la suma de los radios de cada esfera, si esta distancia es menor que el resultado de la suma, entonces se detecta colisión (ver Figura 3.2). Otra ventaja de esta estructura es que la única transformación que debe realizarse sobre la esfera envolvente, son las translaciones del objeto, sin requerir aplicar rotaciones por lo que resulta más rápida que las técnicas AABB u OBB. Sin embargo, se pueden generar reportes erróneos de colisiones si los objetos encerrados no bordean totalmente a sus esferas [2].

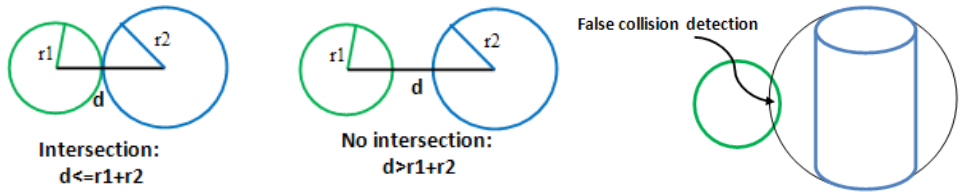

Figura 3.2 Técnica *Bounding Sphere* [2].

3.1.2 Técnica de *Axis Aligned Bounding Box* AABB.

AABB acrónimo ingles de *Axis Aligned Bounding Box* o Cajas Envolventes Alineadas con los Ejes, es un técnica que permite crear una caja alineada con el sistema coordenado de ejes *x*, *y* y *z* alrededor de cada objeto colisionable. Las dimensiones de dicha caja dependen de los valores máximos y mínimos de los bordes del objeto [2].

Aunque pueden existir, la técnica de AABB reduce la aparición de reportes falsos de colisiones con respecto a la técnica SB. El algoritmo es muy fácil de implementar, si se suponen dos objetos A y B encerrado dentro de dos cajas alineadas como en la Figura 3.3, entonces la detección de colisión se realiza comparando las posiciones de los ejes coordenados con respecto al centro del cada objeto, así en una sola dimensión por ejemplo [2]:

1. Si la posición X-máxima de A es menor que la posición X-mínima de B entonces no hay colisión.
2. Si la posición X-máxima de B es menor que la posición X-mínima de A entonces no hay colisión.
3. Se repite el proceso para cada eje.
4. Si ninguna condición es verdadera, entonces se dará la colisión entre los dos objetos.

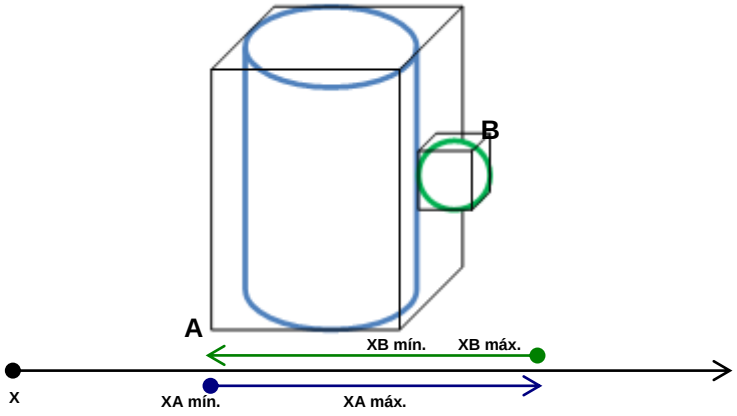

Figura 3.3 Técnica *Axis Aligned Oriented Bounding Box* AABB [2].

Una desventaja de la técnica AABB es que puede crear reportes falsos si los objetos encerrados son delgados y están inclinados, rotados o tienen alguna deformación (ver Figura 3.4), por los que se requiere una actualización regular de la información de las cajas envolventes.

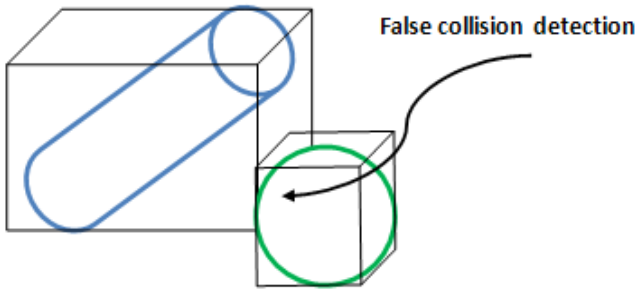

Figura 3.4 Falso Reporte Técnica *Axis Aligned Oriented Bounding Box* AABB [2].

3.1.3 Técnica OBB: *Oriented Bounding Box.*

A través de la técnica OBB, (acrónimo de *Oriented Bounding Box* o caja envolvente orientada) se encierra un modelo geométrico a colisionar, dentro de una caja que está alineada con las dimensiones máximas del objeto (Figura 3.5), de esta forma se calcula la existencia de colisiones con la caja, en lugar de realizar el cálculo con cada punto del objeto. Este tipo de volumen es el indicado sobre todo para objetos que se adapten a formas rectangulares, y aunque exista error en figuras con geometría curvas, BB es el método más eficiente. Para una detección de colisión segura, las trasformaciones de la figura se deben aplicar continuamente a la caja envolvente [2].

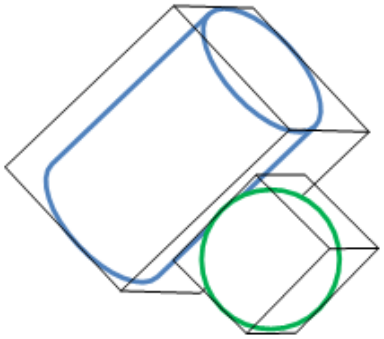

Figura 3.5 OBB *Oriented BoundingBox* [2]

3.2 Bibliotecas para Detección de Colisiones

Se puede definir a un motor físico como un programa por computadora que simula modelos físicos newtonianos, usando variables como masa, velocidad, fricción, peso y resistencia. Este puede simular y predecir efectos bajo diferentes condiciones que se aproximan a lo que ocurre en la realidad.

La principal tarea de un motor físico es realizar la detección de colisiones, resolver colisiones y otras restricciones, proveer la transformada del mundo actualizada para todos los objetos. En esta sección se dará un resumen de algunas bibliotecas estudiadas para la realización de la detección de colisiones.

3.2.1 V- Collide

V-Collide es una biblioteca de detección de colisiones desarrollada por el grupo GAMMA (*Geometric Algorithms for Modeling, Motion, and Animation*) de la Universidad de Carolina del Norte. Esta escrita en C++ y fue diseñada para trabajar en ambientes que contienen gran número de objetos geométricos formados por mallas de triángulos, realizando la detección de colisiones entre parejas de objetos a través de dos etapas [2]:

- Construcción de OBB's (*Hierarachical Oriented Bounded Box*) para cada objeto con el fin de encontrar parejas de triángulos que posiblemente intersecten los OBB's (esto se hace con RAPID).

- Verificar que las parejas de triángulos detectados en la etapa 1 realmente se intercepten.

Estas etapas se encuentran en un componente de V-Collide llamado RAPID, el cual es también una librería de detección de colisiones independiente. Las diferencias entre ambas librerías son las siguientes [2]:

- V-Collide conserva la información acerca de dónde se encuentran los objetos en el ambiente, de tal manera que si éstos no se mueven sus locaciones no tiene porque ser recalculadas con RAPID.

- V-Collide permite la verificación de muchos objetos de forma simultánea y RAPID solamente permite la verificación de dos.

- RAPID reporta qué parejas de triángulos exactamente colisionaron, mientras que V-Collide solo reporta colisión entre objetos.

Se realizaron algunas pruebas con V-Collide en OpenGL, para las cuales fue necesario crear un archivo .obj con la forma del abdomen, obteniendo resultados

satisfactorios, el problema se presentó en el momento de incluir los archivos .obj en Ogre 3D.

Por lo tanto, no se seleccionó esta biblioteca debido a que presenta incompatibilidades con el motor gráfico Ogre 3D en el formato de los archivos, pues esta biblioteca trabaja con archivos .obj y Ogre con archivos .mesh.

3.2.2 Bullet

Bullet Physics es una biblioteca de código abierto para detección de colisiones y dinámica entre cuerpos rígidos y blandos. La biblioteca es gratuita para uso comercial bajo la licencia Zlib y puede ser usada sobre todas las plataformas incluyendo Playstation3, Xbox 360, Wii, PC, Linux, Mac OSX y Iphone [26]. Realiza simulación de cuerpos rígidos y blandos con detección de colisiones continuas y discretas. Entre las formas de colisión se incluyen mallas cóncavas y convexas, mallas de triángulos y todas las primitivas básicas como esferas, conos, cilindros y cajas envolventes. Presenta soporte para cuerpos blandos en ropa, cuerdas y objetos deformables. También cuenta con un amplio conjunto de restricciones para cuerpos blandos y rígidos con límites y motores de restricción [26].

Bullet ha sido diseñada para ser personalizada y modular (ver Figura 3.6). El desarrollador por ejemplo puede usar solamente el componente de detección de colisiones o puede usar el componente de dinámica de cuerpos rígidos sin usar el componente de cuerpos blandos.

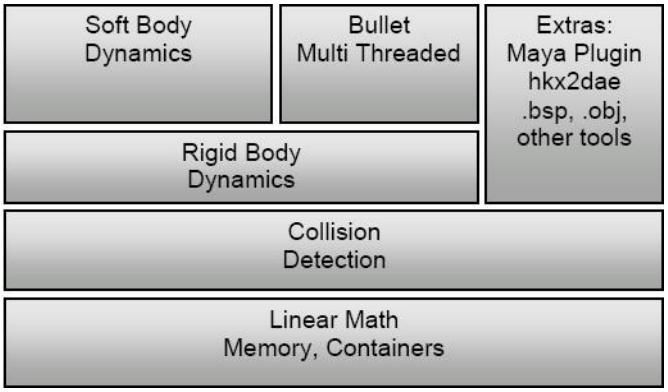

Figura 3.6 Bullet Physics [26].

Para el uso de Bullet con Ogre 3D se estudió Btogre [27] que es una biblioteca software que envuelve las funciones y clases más importantes de Bullet y trabaja en forma de intermediario entre los dos motores.

Debido a la falta de documentación, soporte al programador y problemas encontrados en el desarrollo de una aplicación que permitiera detectar colisiones entre objetos, se descartó el uso de esta biblioteca.

3.2.3 NVIDIA PhysX

Se le da el nombre de PhysX a un chip y a un SDK (Kit de Desarrollo de Software) desarrollado por AGEIA [28] para realizar cálculos físicos complejos. PhysX es un producto comercial pero de uso libre en proyectos no comerciales, está escrito en C++ soportado inicialmente en Windows pero recientemente se ha expandido su portabilidad a Linux.

Es capaz de simular cuerpos rígidos estándar, fluidos dinámicos, ropa, cuerpos blandos, volúmenes, entre otros. El software de PhysX ha sido adoptado para la creación de más de 150 juegos y lo emplean más de 10.000 desarrolladores de todo el mundo. PhysX es utilizado comercialmente por SONY en su consola de juegos PlayStation3.

Para dar un poco de claridad sobre los elementos (clases) más significativos del motor y su funcionamiento se realizó un diagrama de clases (ver Figura 3.7) muy resumido y simplificado, rescatando la clases más significativas, basado en la experiencia adquirida que no pretende ser una formalización de un diagrama de clases para PhysX.

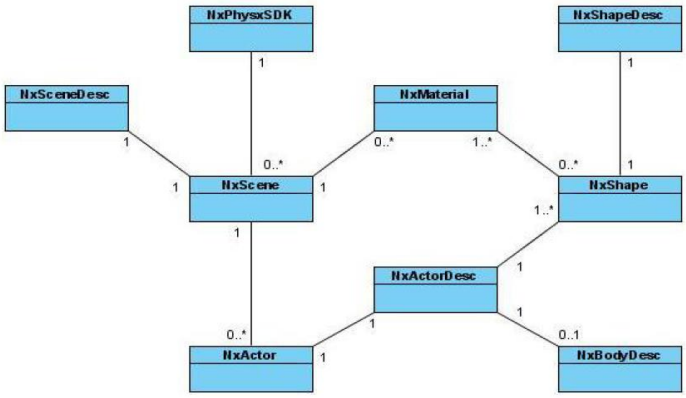

Figura 3.7 Diagrama de clases principales de PhysX [20].

El prefijo “Nx” que se utiliza en todos los nombres de las clases se debe a que anteriormente el SDK era conocido con el nombre de “**N**ovode**x**”, de donde heredó el prefijo.

Al igual que Ogre, PhysX tiene un punto de entrada para inicializar su motor, este punto de entrada es la clase ***NxPhysxSDK***. Lo primero entonces, antes de realizar una simulación, es crear una instancia de la clase NxPhysxSDK para inicializar el sistema, a través de este objeto se pueden ajustar parámetros, por ejemplo dar la opción de visualizar los ejes de un actor (XYZ) para saber cómo está orientado o hacia donde se está aplicado una fuerza. Una vez que el sistema se ha inicializado y ajustado según convenga a la simulación el objeto (instancia) NxPhysxSDK está encargado de crear una instancia de la clase ***NxScene*** [20].

La clase ***NxScene*** en forma similar a Ogre define una escena en donde se encuentran objetos, en este caso **Actores**, en un ambiente de tres dimensiones que permite observar el comportamiento de estos objetos según las condiciones físicas a las que se sometan. Al momento de crear una escena por parte de NxPhysxSDK, es necesario especificar la gravedad que afectará a los objetos en la escena, esto se hace a través de un descriptor de escena que es una instancia de la clase ***NxSceneDesc*.** Los descriptores son muy usados por PhysX y son estructuras que contienen toda la información sobre las características que se desea dar a los objetos al momento de crearlos [20].

La clase ***NxActor*** comprende los elementos básicos de una escena, es decir, las instancias de la clase NxActor, mejor dicho los actores, son los objetos que interactúan unos con otros dentro de la escena. Los actores dentro de una escena son creados por esta misma, es decir, la instancia de la clase NxScene que define la escena. PhysX emplea tres tipos de actores: los estáticos, dinámicos y cinemáticos. Un actor estático como su nombre lo indica no se moverá nunca, puede ser utilizado para representar suelos, montañas, una intersección etc; se considera que estos actores poseen una masa infinita y no son afectados por fuerzas. Un actor dinámico se moverá y actuará bajo la influencia de las distintas situaciones a las que se someta, fuerzas o velocidades, poseen una masa finita. Los actores cinemáticos son objetos estáticos movibles, no se moverán por la aplicación de una fuerza o impacto pero pueden ser posicionados dentro de la escena según convenga. Los objetos dinámicos pueden ser convertidos en cinemáticos y viceversa, no así los estáticos que no pueden ser modificados en tipo. Al igual que con una escena al momento de crear un actor hay que especificar todas las características que se desea tenga este actor (entre ellas su tipo, cuerpo, forma, densidad) esto se hace a través de un *descriptor de actor,* instancia de la clase ***NxActorDesc*** [20].

Un actor está conformado por un cuerpo rígido, lo que PhysX denomina "*Body*", una forma de colisión, "*Shape*", que como su nombre lo indica es la forma que tomará un actor y se comportará en consecuencia. Es decir, si se crea un actor con forma de esfera este actor se verá y comportará como lo hace una esfera, así también si se crea otro actor con forma de cubo (caja) este se verá y comportará como tal. Como se mencionó anteriormente toda esta información está contenida dentro del descriptor de actor, si a un actor no se le asigna un cuerpo rígido (*body*) se considera como de tipo estático. La manera de asignar un cuerpo rígido a un actor es a través de la creación de un *descriptor de cuerpo,* esto es, la creación de una instancia de la clase ***NxBodyDesc*** [20].

La clase ***NxShape*** define las formas con las que trabaja PhysX, éstas pueden ser: planos, cajas, esferas, capsulas, ruedas, triángulo *mesh*. Para crear cualquiera de las formas anteriores se deben especificar características como dimensiones y materiales entre otras, esto se hace, al igual como en los casos anteriores, a través de descriptores y en este caso *descriptores de forma* que son instancias de la clase ***NxShapeDesc.*** Un actor puede tener varias formas asociadas como es el caso de un vehículo que en sí corresponde a un actor pero con una forma

asociada para representar el chasis y otras cuatro formas para representar cada una, una rueda. La forma “triángulo *mesh*” da la posibilidad de crear formas más complejas, especialmente en herramientas modeladoras desde donde posteriormente se pueden exportar a un formato que puede ser leído por PhysX [20].

La clase ***NxMaterial***. A diferencia de Ogre en donde los materiales están más relacionados con la apariencia, en PhysX son considerados desde el punto de vista físico.

Los materiales por lo tanto son la sustancia física de la que están hechos los objetos (actores o más directamente las formas), definen propiedades internas como restitución y propiedades de la superficie del objeto como fricciones. Los materiales son creados por el objeto escena y es necesario ajustar sus propiedades dando valores a su restitución, fricción estática y fricción dinámica. Dar un valor alto de restitución, por ejemplo el máximo que es 1, significa que en un impacto el objeto perderá poca energía lo que llevará a que este tenga un rebote amplio, en el caso contrario si se le asigna un coeficiente de restitución bajo, por ejemplo de 0,15 el objeto perderá mucha energía en el impacto y rebotará poco. Si un material se ajusta con un coeficiente de fricción estática elevado provocará que el objeto tenga dificultades en desplazarse al intentar salir del reposo, esta dificultad disminuirá si disminuye el coeficiente de fricción. Un coeficiente de fricción dinámica pequeño permitirá a un objeto en movimiento deslizarse suavemente mientras que uno elevado tenderá a detenerlo de manera más rápida o más brusca. Una forma puede usar un material creado por la escena así como también puede dejar de usarlo y cambiarlo por otro en cualquier momento [20].

La Figura 3.8 muestra el entorno de trabajo utilizado por PhysX en los tutoriales que tiene disponibles para el aprendizaje del motor. En la imagen se puede apreciar una escena donde un actor con forma de esfera cae por acción de la fuerza de gravedad, golpeando una pila de actores con forma de cajas.

Figura 3.8 Entorno de trabajo de PhysX.

De los motores físicos estudiados que se integran a aplicaciones desarrolladas con Ogre destaca por su rapidez y robustez, según la opinión de los propios usuarios recogida en los foros de Ogre [29], PhysX.
El hecho de seleccionar PhysX como motor físico para el desarrollo del simulador se basó fundamentalmente en la compatibilidad con Ogre 3D y la posibilidad de trabajar con NxOgre, ya que con éste se logró el primer acercamiento a la simulación de la física por recomendación de usuarios expertos de Ogre y disponibilidad de soporte en el aprendizaje, a través de tutoriales y un foro muy activo y cooperativo.

3.3 NxOgre

Para usar el motor físico PhysX con Ogre 3D se utilizó un "*wrapper*" físico llamado NxOgre, es una biblioteca de software que envuelve todas las funciones y clases de PhysX y actúa como intermediario. Es una biblioteca de código abierto sobre la licencia LGPL. Existen dos versiones de NxOgre actualmente: BloodyMess y Bleeding, siendo la primera la última versión de esta biblioteca. Aunque esta librería tenga Ogre en su nombre podrá funcionar con cualquier motor de renderizado 3D o 2D.

NxOgre es un envoltorio creado por Robin Southern que integra a Ogre con PhysX. De esta manera al trabajar con NxOgre es posible acceder, a través de él, a ambos motores, gráfico y físico, dando la posibilidad de desarrollar aplicaciones con todo el potencial que posee cada uno. Al igual que con Ogre y PhysX, NxOgre está escrito en lenguaje de programación C++, es orientado a objetos y tiene licencia de código abierto pero está limitada por el uso de PhysX que como se mencionó anteriormente es pagada pero de libre uso en proyectos no comerciales [20].

NxOgre posee muchas clases que envuelven a las clases definidas para Ogre y PhysX. Con el objetivo de dar una visión más clara de cómo trabaja NxOgre se desarrolló un diagrama de clases (ver Figura 3.9) con las características más relevantes y significativas que permitan entender al envoltorio.

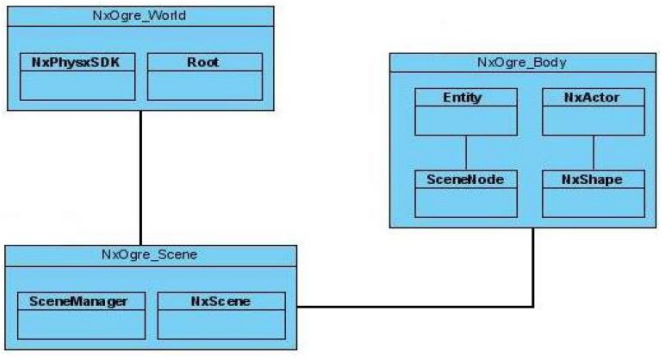

Figura 3.9 Diagrama de clases de NxOgre [20].

Como se mencionó con anterioridad Ogre y PhysX necesitan partir inicializando sus sistemas, esto lo consiguen mediante la clase *Root* y NxPhysxSDK respectivamente.

NxOgre realiza este trabajo a través de la clase ***NxOgre_World***, el objeto NxOgre_World toma como parámetro el objeto Root de Ogre ya inicializado y se encarga de inicializar el SDK PhysX mediante la creación de un objeto NxPhysxSDK. A través del objeto NxOgre_World se puede acceder a todos los métodos disponibles para los objetos de las clases inicializadoras de ambos motores, además el objeto NxOgre_World es el responsable de crear la escena de NxOgre [20].

La creación de una instancia de la clase NxOgre_Scene lleva implícita la creación de una escena tanto para Ogre, mediante un objeto de la clase *SceneManager*, como para PhysX, con la creación de un objeto de la clase NxScene. Al igual que ocurre con el objeto NxOgre_World, la escena de NxOgre otorga acceso a todos los métodos disponibles para las instancias de SceneManager y NxScene, permitiendo obtener toda la funcionalidad de las escenas de Ogre y PhysX. Como se puede apreciar hasta aquí el trabajo de los motores se realiza en paralelo, siendo NxOgre el nexo entre los dos.

Ogre define los objetos dentro de su escena como “Entidades”, PhysX por su lado los llama “Actores”. NxOgre a través de la clase ***NxOgre_Body*** integra los dos conceptos de objetos dentro de una escena, tanto el de Ogre como el de PhysX, y les da el nombre de “*Bodies*” o “*Body*”. Cuando NxOgre crea un *Body* mediante y dentro de su escena está creando al mismo tiempo un actor para PhysX y una entidad para Ogre, así logra realizar las simulaciones físicas con el actor y representar esto gráficamente a través de la entidad creada para Ogre, a los que se puede llamar también “Modelo de colisión” y “Modelo Gráfico” respectivamente. En donde el modelo de colisión corresponde a la forma (*shape*) sobre la cual se realizan los caculos físicos y el modelo gráfico corresponde a la entidad o *mesh* sobre la cual se basa la representación gráfica del objeto [20].

Mediante un objeto NxOgre_Body se puede tener acceso a todos los métodos de la clase Entity y todos los métodos de la clase NxActor. También al crear un objeto NxOgre_Body se enlaza la entidad creada para Ogre a un nodo de la clase *SceneNode* que permite tener control sobre la entidad. NxOgre utiliza, a través del uso de su escena (clase NxOgre_Scene), la escena de la clase *SceneManager* para trabajar con las entidades y la escena de la clase NxScene para trabajar con los actores. Cada movimiento realizado por un actor (modelo de colisión) es replicado por la entidad (*mesh*) de esta manera se puede observar gráficamente a través de Ogre 3d el comportamiento de los objetos dentro de una simulación física realizada por PhysX. Por último, si se crea un “*Body*” con forma de esfera (shape) y una apariencia gráfica (*entity* o *mesh*) de una jirafa, este *Body* no se comportará como jirafa sino que como esfera y lo más probable que la jirafa ruede antes que caminar. El desarrollo del simulador se realizará sobre la base de las clases definidas por NxOgre, de esta manera se logra integrar al proyecto el motor

gráfico Ogre3d y el motor físico PhysX. Cualquier funcionalidad no cubierta por NxOgre tiene la facilidad de ser salvada interactuando directamente con cualquiera de los dos motores, ésta es una ventaja que posee NxOgre [20].

3.4 Detección de Colisiones y Realimentación de Fuerza

A continuación se describirá cómo se realizó la detección de colisiones entre el efector final del robot y el abdomen simulado del paciente con las herramientas software mencionadas anteriormente.

La versión del SDK PhysX utilizado para el desarrollo del proyecto es la 2.8.3, la última versión lanzada para el SDK corresponde a la 2.8.4. Se utiliza la versión 2.8.3 por que cuando se comenzó el entrenamiento con la herramienta ésta correspondía a la última versión lanzada por Ageia y la versión de NxOgre es BloddyMess 1.5.5.

El diagrama de flujo de la Figura 3.10 presenta los pasos realizados para lograr la detección de colisiones entre el efector final y el abdomen simulado:

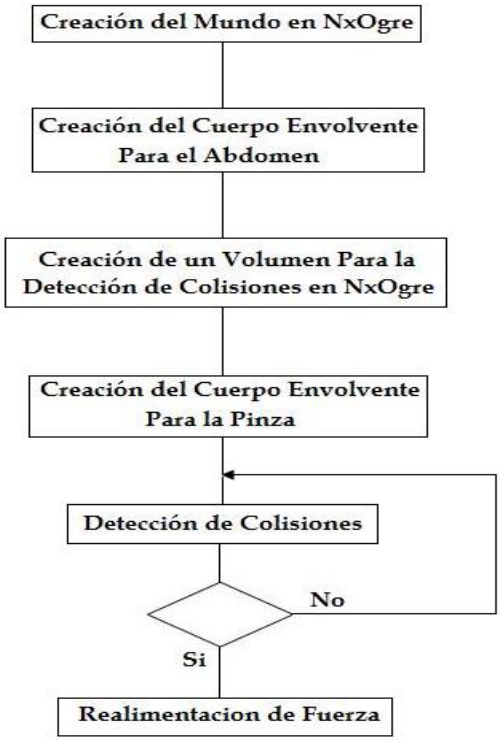

Figura 3.10 Diagrama de Flujo para Detección de Colisiones

3.4.1 Creación del mundo en NxOgre

Para trabajar con NxOgre como se mencionó anteriormente lo primero que se debe crear es el mundo, después la escena junto a su descripción, algunos valores físicos necesarios para la simulación, el sistema de renderizado y el controlador de tiempo.

```
// Create the world
mWorld = NxOgre::World::createWorld();
 // Create scene description
NxOgre::SceneDescription sceneDesc;
sceneDesc.mGravity = NxOgre::Vec3(0, 0, 0);
sceneDesc.mName = "DemoScene";

// Create scene
mScene = mWorld->createScene(sceneDesc);

// Set some physical scene values
mScene->getMaterial(0)->setStaticFriction(0.5);
mScene->getMaterial(0)->setDynamicFriction(0.5);
mScene->getMaterial(0)->setRestitution(0.1);

// Create render system
mRenderSystem = new OGRE3DRenderSystem(mScene);

//Create time controller
mTimeController = NxOgre::TimeController::getSingleton();
```

3.4.2 Creación del Cuerpo Envolvente Para el Abdomen

Inicialmente se trabajó en la colisión entre el abdomen simulado del paciente y un objeto cualquiera. Para esto se estudiaron los tutoriales de BloddyMess [30] para la creación de formas físicas complejas en las aplicaciones de detección de colisiones.

Debido a la forma del abdomen fue necesario construir un cuerpo a base de triángulos que permitiera detectar colisión en todos los puntos del abdomen.

El abdomen utilizado en esta aplicación fue construido en Solid Edge y se exportó como un archivo .stl, luego se importó a Blender [31] para exportarlo como un archivo .mesh, esto debido a que Ogre 3D trabaja con este tipo de archivos. Un *mesh* es esencialmente un polígono complejo en 3D, representado por un conjunto de vértices y un cuerpo triangulizado permite representar perfectamente la forma física de un *mesh*.

Para construir el cuerpo triangulizado fue necesario usar la herramienta Flour [32], que permite convertir de archivos .mesh a archivos .nxs con los cuales se representan los cuerpos triangulizados. El cuerpo generado por Flour es totalmente igual en forma y tamaño al que contiene el archivo .mesh, pero es necesario para trabajar con NxOgre y PhysX.

Una vez creado el cuerpo con Flour, se debe cargar en Ogre para trabajar con él. Las siguientes líneas de código permiten cargar el cuerpo:

```
NxOgre::ResourceSystem::getSingleton()->openArchive("media", "file:C:/OgreSDK/media");
NxOgre::Mesh* triangleMesh = NxOgre::MeshManager::getSingleton()->load("media:Mesh.nxs");
```

Ahora se debe crear un "*SceneGeometry*" que permite generar el cuerpo triangulizado en la escena y para poder visualizarlo se crea una entidad y se les asigna la misma posición en la escena:

```
NxOgre::TriangleGeometry* triangleGeometry = new NxOgre::TriangleGeometry(triangleMesh);
Entity *Abdomen = mSceneMgr->createEntity("Abdomen", "Mesh.mesh");
Abdomen ->setMaterialName("MatAbdomen");
SceneNode *nodoAbdomen =  mSceneMgr->getRootSceneNode()->createChildSceneNode(Vector3( 10.475, -
2.65, 8.8));
nodoAbdomen->attachObject(Abdomen);
mScene->createSceneGeometry(triangleGeometry, NxOgre::Matrix44(NxOgre::Vec3(10.475, -2.65, 8.8)));
```

3.4.3 Creación de un Volumen Para la Detección de Colisiones

Estudiando los tutoriales de NxOgre, el autor plantea un método para la detección de colisiones que consiste en crear un volumen que tenga la misma forma del abdomen, el cual permitiría detectar la colisión en cualquier punto del abdomen. Un volumen es un objeto físico invisible de una forma arbitraria, no se mueve ni puede ser visto. Este volumen se usa en este caso como un gatillo, tan pronto como un objeto entre en él, esté completamente dentro o salga de él, un evento especial es disparado y puede servir para realizar una determinada acción dentro del programa, en este caso la detección de colisión.

Es necesario hacer que la clase principal del programa herede de NxOgre::Callback para que sea capaz de manejar los eventos que se generen cuando un objeto entre en contacto con el volumen creado.

Donde se creó la escena quirúrgica, ahora se crea entonces un volumen que tiene las siguientes características:

- La forma que es la misma del abdomen simulado *triangleGeometry.*
- La posición que es la misma que tiene el abdomen.
- La clase *Callback* de la cual ya se indicó la herencia.
- El comportamiento, que para este caso el método llamado es *OnVolumeEvent()*, el cual reaccionará cuando un cuerpo entre o salga del volumen creado.

Las siguientes líneas de código permiten ver la creación del volumen:

```
mVolume =  mScene->createVolume(triangleGeometry, NxOgre::Matrix44(NxOgre::Vec3(10.475, -2.65, 8.8)), this,
NxOgre::Enums::VolumeCollisionType_All);
```

Lo último que se debe hacer para detectar la colisión es crear la función que será ejecutada cuando un objeto entre en contacto con el volumen.

```
void onVolumeEvent(NxOgre::Volume* volume, NxOgre::Shape* volumeShape, NxOgre::RigidBody* rigidBody,
NxOgre::Shape* rigidBodyShape, unsigned int collisionEvent)
{
if(collisionEvent == NxOgre::Enums::VolumeCollisionType_OnEnter)
   {
```

```
                // Dentro de esta condición se programa las acciones a tomar cuando se detecte una colisión.
        }
    }
```

Se creó un objeto con forma de barril con una caja envolvente para probar la colision entre dos objetos en el mundo virtual y se obtuvieron los resultados que se muestran en la
Figura 3.11:

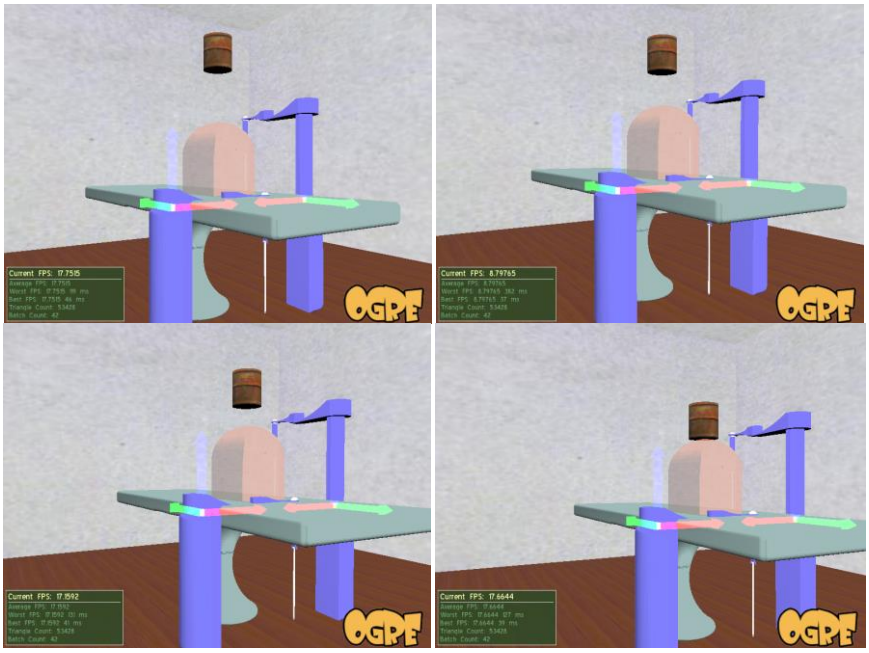

Figura 3.11 Colisión entre dos objetos.

3.4.4 Creación del Cuerpo Envolvente Para la Pinza

Después de la creación del volumen para la detección de colisiones, se creó un cuerpo envolvente en forma de esfera para que recubriera a la pinza y así poder realizar la detección de colisiones con el abdomen simulado del paciente.

Con las siguientes líneas de código se muestra la creación de la esfera:

```
Body = mRenderSystem->createBody(new NxOgre::Sphere(0.18),NxOgre::Vec3(10.475, 2, 8.8), "pinza.mesh");
```

Se le debe asignar a este cuerpo la posición del cursor háptico para que siga los movimientos de la pinza.

3.4.5 Detección de Colisiones

Para realizar la detección de colisiones, como se mencionó anteriormente, se hace uso de un volumen y del método *OnVolumenEvent()*, dentro de este se utiliza la condición `if(collisionEvent == NxOgre::Enums::VolumeCollisionType_OnEnter)` la cual permite

saber si algún cuerpo entró en contacto con el volumen, en este caso nos indica si el efector final del robot entró en contacto con el abdomen simulado.

Si esta condición se cumple se le asigna a una bandera verdadero para saber que se detectó una colisión, esto con el fin de poder realizar como siguiente paso la realimentación de fuerza al efector final.
Una vez se detecta la colisión entre el efector final y el abdomen, se le muestra al usuario el mensaje en pantalla "--- COLISION DETECTADA ---" (Figura 3.12), indicandole que esta tocando las paredes del abdomen.

Figura 3.12 Mensaje en pantalla.

3.4.5.1 Resultados para la Detección de Colisiones

A continuación en la Figura 3.13 se muestran los resultados obtenidos en la detección de colisiones entre el efector final y el abdomen simulado del paciente, el ovalo rojo en la figura resalta el mensaje que indica la detección de la colisión:

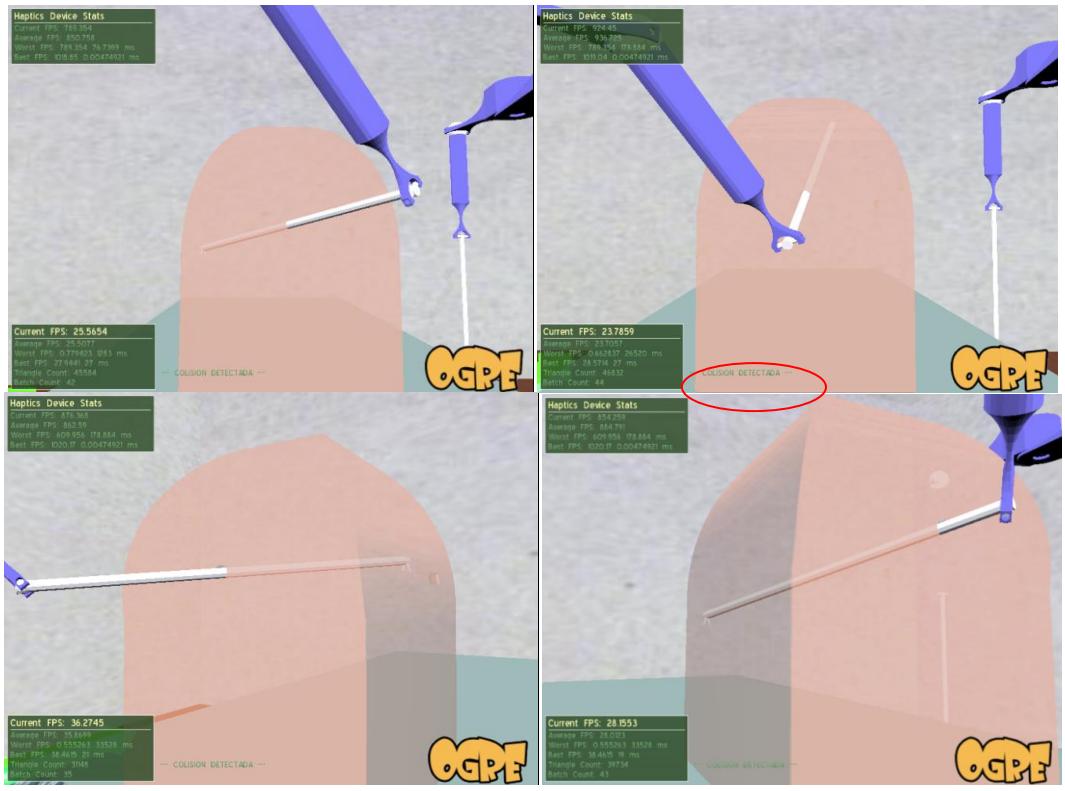

Figura 3.13 Detección de colisión.

En la Figura 3.14 se muestra la detección de colisión vista desde el interior del abdomen del paciente:

Figura 3.14 Detección de colisión dentro del abdomen.

3.4.6 Realimentación de Fuerza

Cuando se detecta una colisión se le envía al dispositivo háptico una fuerza que le indica al usuario que está tocando una de las paredes del abdomen.

En este caso se utilizó un algoritmo de fuerza llamado *Spring* o Resorte en OgreHaptics [33] correspondiente a la ecuación F=-KX, la ley de Hooke[8], donde X corresponde a la diferencia entre la posición actual y la posición anterior del dispositivo háptico, F a la fuerza que debe ser generada por el dispositivo y K la constante de fuerza.

En las siguientes líneas de código se puede apreciar el algoritmo y las funciones utilizadas para llevar a cabo la realimentación de fuerza al dispositivo háptico. Lo primero que se debe hacer es obtener la diferencia entre la posición antes y después de la colisión y se almacena este valor en un vector para después multiplicarlo por una constante de fuerza.

```
Vector3 X=pactual-pante;
Vector3 F=-100*X;

force[0]=F.x;
force[1]=F.y;
force[2]=F.z;
```

Para aplicarle fuerzas al dispositivo háptico se estudiaron dos funciones de dos bibliotecas diferentes recomendadas por expertos en Ogre. La primera corresponde a una función de OgreHaptics llamada *_applyForces()* que aplica a la interfaz háptica una fuerza en coordenadas del dispositivo. Y la segunda corresponde a una función de la biblioteca háptica HDAL llamada

[8] Cuando un resorte es deformado ejerce una fuerza opuesta a la deformación que es proporcional a la magnitud de la deformación sobre el resorte.

hdlSetToolForce() la cual ajusta la fuerza para ser generada por el dispositivo háptico en Newtons.

```
mDevice->_applyForces(F, Vector3::ZERO);
hdlSetToolForce(force);
```

De las dos funciones mencionadas la que arrojó mejores resultados fue la última, ya que se obtiene una mayor realimentación de fuerza al tocar las paredes del abdomen simulado del paciente.

El código detallado de la aplicación se encuentra en el Anexo B.

4. Conclusiones

En este proyecto se realizó el posicionamiento de un robot asistente para operaciones de laparoscopia llamado LapBot, por medio de la interfaz háptica Novint Falcon, haciendo un estudio previo de las bibliotecas hápticas que permiten la comunicación entre el dispositivo háptico y el motor de renderizado grafico Ogre 3D.

De este estudio se concluyó que OgreHaptics es una de las mejores opciones para trabajar con Ogre 3D por su compatibilidad con el motor grafico y con Novint Falcon, lo que permite al usuario manipular los entornos virtuales en 3D usando retroalimentación de fuerza. Está diseñado para hacer más fácil y más intuitivo a los desarrolladores el producir demos y juegos que utilizan gráficos 3D y dispositivos hápticos.

Gracias a la exploración de la biblioteca OgreHaptics se desarrolló un programa donde se obtuvo el posicionamiento de un objeto en el ambiente virtual de Ogre 3D haciendo uso del dispositivo háptico.

Posteriormente se realizó un estudio de los motores físicos disponibles para simular la detección de colisiones entre dos objetos virtuales. De este estudio se obtuvo que la mejor opción para este proyecto; es el motor físico NVIDIA PhysX. El hecho de seleccionar PhysX como motor físico para el desarrollo del proyecto se basó fundamentalmente en la compatibilidad con Ogre 3D y la posibilidad de trabajar con el *wrapper* NxOgre, ya que con éste se logró el primer acercamiento a la simulación de la física por recomendación de usuarios expertos de Ogre y disponibilidad de soporte en el aprendizaje, a través de tutoriales y un foro muy activo y cooperativo.

NxOgre se constituye en una gran alternativa para la detección de colisiones, pues da al desarrollador la posibilidad de trabajar con volúmenes envolventes a base de triángulos para representar figuras complejas, permitiendo así la detección de colisión en todos los puntos de la figura. Además, permite detectar colisión cuando un objeto entra, sale y/o está dentro del volumen.

Por último se estudiaron dos métodos de las bibliotecas OgreHaptics y HDAL, para realizar la realimentación de fuerza a la interfaz háptica, logrando mejores

resultados con la función de HDAL, ya que se obtiene una mayor realimentación de fuerza en el dispositivo.

Como trabajos futuros se plantea el posicionamiento del segundo brazo del robot con un nuevo dispositivo háptico, que incluya detección de colisiones y realimentación de fuerza. Además de la construcción de los órganos virtuales que permitan dar un mayor realismo al simulador. También se busca implementar la deformación de objetos inicialmente para el abdomen y luego para todos los órganos que se incluyan en la simulación.

5. Discusión

En este último capítulo se dan algunas recomendaciones que servirán como soporte para continuar con los trabajos futuros propuestos y que le permitirán a los estudiantes minimizar el tiempo de trabajo e investigación. Las recomendaciones propuestas están hechas sobre las herramientas software seleccionadas para el desarrollo de este proyecto.

a. Conexión de otro dispositivo háptico:

Si se desea trabajar con el dispositivo háptico Phantom se debe descomentar (borrar #) en el archivo HapticsPlugin.cfg la línea que permite trabajar con OpenHaptics para el manejo de Phantom:

Plugin = HapticsRenderSystem_OpenHaptics

Si se desea añadir la interfaz háptica Phantom además de Novint Falcon, por ejemplo para el posicionamiento del segundo brazo en la aplicación, se debe crear una nueva instancia de la clase device que le permita al sistema reconocer el nuevo dispositivo y así se pueda trabajar con esta interfaz. A continuación se presentan unas líneas de código que pueden ser probadas en la aplicación. No olvidar activar los plugins correspondientes a cada interfaz.

```
DeviceInitInfo initInfo;

initInfo.api = "Falcon HapticsRenderSystem";
initInfo.initName = "DEFAULT";
Device* falconDevice =  System::getSingleton().createDevice("Falcon", initInfo);

initInfo.api = "OpenHaptics HapticsRenderSystem";
initInfo.initName = "DefaultPHANToM";
Device* phantomDevice = System::getSingleton().createDevice("PHANToM", initInfo);

System::getSingleton().startSchedulers();
```

b. Creación de cuerpos envolventes compuestos para detección de colisiones con órganos:

Si se desea crear un cuerpo envolvente para las demás articulaciones de LapBot se recomienda utilizar directamente PhysX, dado que el wrapper NxOgre solo envuelve las clases principales del motor físico y no la totalidad de las mismas.

Al instalar el motor físico Physx se crea un directorio “C:\Archivos de programa\NVIDIA Corporation\NVIDIA PhysX SDK\v2.8.3” en el que se encuentra la documentación necesaria para aprender a manejar esta herramienta.

Para crear un cuerpo envolvente que permita detectar la colisión con la pinza y las articulaciones del robot que quedan dentro del abdomen, se debe estudiar el programa de entrenamiento llamado Lesson103, que viene con el motor físico PhysX, donde se explica al programador de que manera puede crear formas de cuerpos envolventes compuestas, lo cual sería muy útil para el caso del brazo de LapBot.

Después es probable que se desee incluir los órganos que van dentro del abdomen, para lo cual se recomienda estudiar los programas de entrenamiento desde Lesson1101 a Lesson1106 que permiten crear cuerpos blandos y los métodos que ofrece el motor físico PhysX de detección de colisiones usando objetos deformables.

En especial se aconseja estudiar un programa de entrenamiento muy interesante, Lesson115 que permite crear una escena en donde se pueden generar colisiones entre cuerpos blandos como los órganos y cuerpos rígidos como el cuerpo del robot que está dentro del abdomen.

La implementación de estos tutoriales en la aplicación actual debe hacerse en el archivo Escena.h para la creación de los cuerpos envolventes y en caso de que el tutorial requiera la actualización de una posición o un cambio en el cuerpo envolvente, este se debe hacer en el archivo Iteracion.h que es el archivo que se está corriendo cada frame.

c. Desconexión de la interfaz háptica Novint Falcon:

El dispositivo háptico no se puede desconectar de la aplicación pues ésta dejaría de funcionar dado que lo primero que se ejecuta es Ogrehaptics que es la biblioteca que maneja las funciones de la interfaz.

Para que el programa funcione sin una interfaz háptica es necesario crear una nueva clase que no herede los métodos de esta biblioteca.

Lo que se puede hacer en caso de no contar con el dispositivo Falcon es conectar la interfaz háptica Phantom y descomentar la línea 291 donde se le envía la orden para que genere fuerza con la función *applyForces*.

d. Realimentación de fuerza:

Para mejorar la realimentación de fuerza que se le envía al usuario se recomienda crear un algoritmo que permita reconocer el eje en el cual se ha detectado una colisión y en ese mismo eje enviar la fuerza. También se aconseja estudiar los *plugins* de OgreHaptics para aplicar efectos de fuerza los cuales se pueden encontrar en el manual que se instala con la biblioteca.

e. Inclusión de los modelos dinámicos del robot en la aplicación

Para la inclusión de los modelos dinámicos de LapBot es necesario crear un programa que realice lo mismo que hace simulink en Matlab pero en C++, es decir:

- Leer las variables articulares que genera el MGI
- Implementar el programa 'trocar' en C++ para que calcule los valores de las variables pasivas, sus velocidades y aceleraciones.
- Con estos datos se implementa el control PD que genera una señal de corrección que debe meterse en el MDI, como aparece en el subsystem de simulink.
- Implementar los modelos dinámicos en C++,
- Y garantizar que la realimentación funcione en C++.

Para este trabajo se deben tener en cuenta los siguientes archivos de matlab que se adjuntan con la aplicación:

- Lapbot2DinCart_mgi_simple.mdl que es el archivo de simulink.
- trocar_opt.m para cálculo de variables.
- MDD.m y MDI.m modelos dinámicos del robot.

f. Conversión de archivos de Matlab a C++

Para este caso se estudiaron dos herramientas: Matlab Compiler y MATLAB C++ Math Library esta última se descartó dado que en esta biblioteca no se pueden implementar funciones de simulink.

Matlab Compiler permite realizar una enorme variedad de conversiones de archivos con 3 objetivos principales:

- Integrar instrucciones Matlab en otros lenguajes como C o Basic, facilitando de esta forma enormemente la codificación de complejas operaciones matemáticas, gráficas, etc.
- Crear bibliotecas de funciones Matlab que puedan ser llamadas desde cualquier aplicación.
- Independizar el código compilado de Matlab del entorno. Es decir, prescindir de la aplicación Matlab a la hora de ejecutar su código, generando aplicaciones independientes (*Stand alone applications*).

Para poder convertir estos archivos se debe tener instalado un compilador de C++ (como Visual Studio) y Matlab. Se debe revisar si está instalado Matlab Compiler, en este caso se hicieron pruebas con Compiler version: 4.11 (MatlabR2009b):

El primer paso al abrir matlab es ir a la ventana de comandos y verificar que matlaba reconozca el compilador de c++ que se tenga instalado en el equipo. Se debe escribir el comando mbuild –setup con el cual se podrá saber que compiladores reconoce el computador. Una vez lo haya identificado se debe ir a inicio->Matlab->Matlab Compiler->Deployment Tool (Figura 5.1).

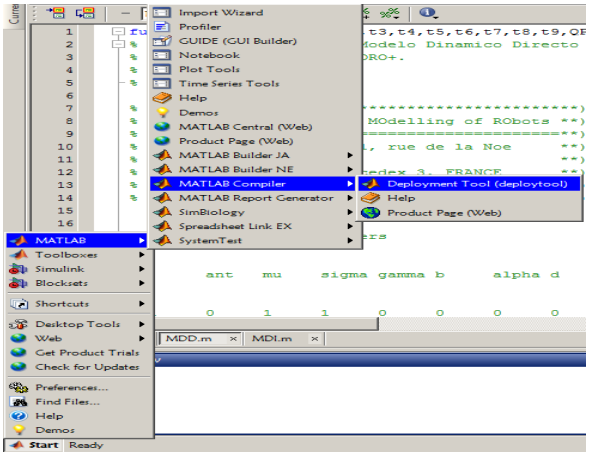

Figura 5.1 Matlab Compiler

Después al abrir esta herramienta se observa una ventana donde se puede escoger el tipo de aplicación que se desea crear, en la Figura 5.2 se puede observar las diferentes clases de aplicaciones que la herramienta ofrece para convertir:

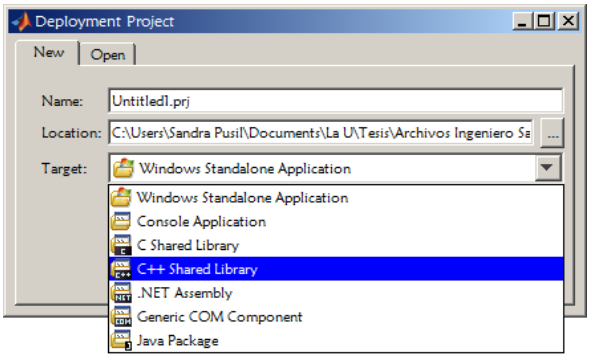

Figura 5.2 Herramienta de conversión

Para hacer esta prueba se seleccionó C++ Shared Library dado que se necesitaba convertir el código en Matlab a c++, una vez se escoge la herramienta, se abre una ventana de Matlab Compiler para la conversión (Figura 5.3):

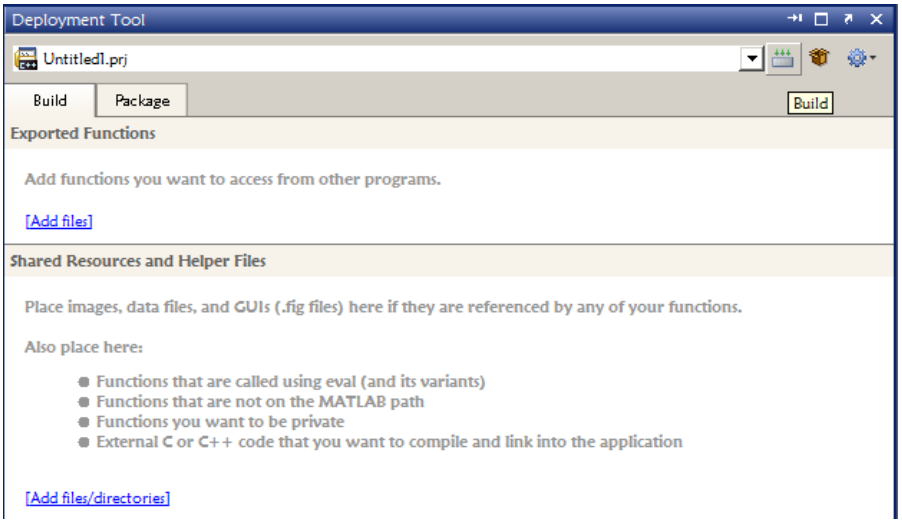

Figura 5.3 Construcción de las aplicaciones

En la pestaña Build en Exported Functions se deben añadir los archivos.m que se deseen convertir a c++ y una vez se carguen estos archivos se da click en el icono Build.

Esta herramienta generara diferentes archivos que pueden ser usados en el compilador de c++ sin necesidad de matlab. Los archivos que se generan son:

- readme.txt
- Untitled.dll
- Untitled.h
- Untitled.lib

Desafortunadamente el Compilador de Matlab no soporta la conversión de archivos de Simulink a código en C++ ni a ningún otro lenguaje de programación ni tipo de aplicación. Para realizar esta tarea se aconseja estudiar la herramienta llamada Real Time WorkShop de Matlab, esta se puede encontrar abriendo el archivo de simulink, una vez abierto en el menú, se abre la pestaña Herramientas y se hace clik en Real Time WorkShop, se recomienda revisar la ayuda de la herramienta para saber la configuración que se debe realizar antes de compilar cualquier código.

Referencias Bibliográficas

[1] UNIVERSIDAD EAFIT. “Simulador de Cirugías Mínimamente Invasivas”. Documento en línea disponible en http://www1.eafit.edu.co/rvirtual/html/simulador.html. Consultado: enero de 2010.

[2] PINTO, M. “Análisis e Implementación de una Interfaz Háptica en Entornos Virtuales”. Tesis de Maestría, Universidad Nacional de Colombia, Bogotá, Colombia. 2009

[3] SAN MARTIN, J. “Aportaciones al Diseño Mecánico de los Entrenadores Basados en Realidad Virtual”. Tesis doctoral, Universidad Rey Juan Carlos, Madrid, España. Marzo 2007

[4] SALINAS, S. “Modelado y Simulación en 3D y Control de un Robot para Cirugía Laparoscópica”. Tesis de Maestría, Universidad del Cauca, Popayán, Colombia. 2009.

[5] IVORRA, D. “Simulador Háptico para Entrenamiento de Técnicas de Laparoscopia”. Tesis de Pregrado, Universidad Miguel Hernández de Elche, Elche, España. 2008.

[6] S. SCHWARTZ y J. HUNTER, *Principios de Cirugía,* 7a ed, Vol. II. México: McGrawHill Interamericana, 1999.

[7] A. CUSCHIERI, “The Spectrum of Laparoscopic Surgery”, *World Journal Surgery,* vol. 16, pp. 1089-1097, 1992.

[8] VIVAS, A. “Aplicaciones de la Robótica al Campo de la Medicina”. Revista Pulsos, No. 9, pp. 32 - 38, 2007.

[9] SIMBIONIX: Lap Mentor http://www.simbionix.com/LAP_Mentor.html. Consultado: enero de 2010.

[10] IMMERSION CORPORATION. “Laparoscopy VR Surgical Simulator”. Documento pdf en linea disponible en http://www.immersion.com/docs/Medical/laparoscopy/LaparoscopyVR_jan10-v3.pdf. Consultado: enero de 2010.

[11] THE UNIVERSITY OF MISSISSIPPI MEDICAL CENTER. “Virtual Laparoscopy Simulator (LapSim)” http://surgery.umc.edu/facultystaff/learning/SSC/virtual.html. Consultado: enero de 2010.

[12] Cirugía Robótica y Simuladores. http://www.depeca.uah.es/docencia/doctorado/cursos04_05/83190/Documentos/CirugiaRoboticaMI-Simuladores.pdf. Consultado: enero de 2010.

[13] The world-leading ProMIS Surgical Simulator http://www.haptica.com/index.htm. Consultado: enero de 2010.

[14] MONSERRAT, C., ALCAÑIZ, M., MEIER, U., POZA,J., LIZANDRA, C. y GRAU, V. “Simulador para el Entrenamiento en Cirugías Avanzadas”. MedICLab/DSIC, Universidad Politécnica de Valencia, Valencia, España. 2002.

[15] NIEMEYER, G., KUCHENBECKER, K., BONNEAU, R., MITRA, P., REID, A., FIENE, J. y WELDON, G. “THUMP: An Inmmersive Haptic Console for Surgical Simulation and Training”. 2001.

[16] URBINA, B., COTO, E., RODRÍGUEZ, O., MIQUILARENA, R. y CERROLAZA, M. “A Virtual System for Laparoscopic Surgery Training”. Memorias de la II Conferencia Internacional de Bioingeniería Computacional. Lisbon, Portugal, septiembre 14 – 16, 2005.

[17] HERRERA, O., ESPITIA, R., ZAYED, G., FIGUEROA, P., TORRES, J. y GARCIA, A. “Desarrollo de un Sistema de Entrenamiento Médico para Artroscopia de Rodilla”. Revista Colombiana de Ortopedia y Traumatología. Agosto 2008

[18] MARTIN, C. “Interfaces Hápticos. Aplicación en Entornos Virtuales”. XVI Congreso Internacional de Ingeniería Grafica, España. 2003

[19] NOVINT TECHNOLOGIES. http://home.novint.com/novint/novint.php. Consultado: enero de 2010.

[20] ALMONACID, O. “Simulación Digital de Tráfico para Intersecciones Señalizadas por Semáforo, Bajo Ambiente Tridimensional”. Tesis de pregrado, Universidad del Bío-Bío, Chile. Septiembre de 2007.

[21] OGRE 3D.www.ogre3d.org. Consultado: enero de 2010.

[22] NOVINT TECHNOLOGIES. “HDAL_Programmers Guide”. Documento pdf en línea disponible en dec-a-cards.googlecode.com/files/HDAL_ProgrammersGuide.pdf. Consultado: enero de 2010.

[23] LIBNIFALCON - Open source driver for the Novint Falcon. http://qdot.github.com/libnifalcon/. Consultado: enero de 2010.

[24] H3D.org - Open Source Haptics. http://www.h3dapi.org/. Consultado: enero de 2010.

[25] OGREHAPTICS. http://ogrehaptics.sourceforge.net/. Consultado: enero de 2010.

[26] BULLET PHYSICS. http://bulletphysics.org/wordpress/. Consultado: julio de 2010.

[27] BTOGRE. http://www.ogre3d.org/forums/viewtopic.php?f=5&t=46856. Consultado: julio de 2010.

[28] AGEIA. http://www.nvidia.com/object/physx_new.html. Consultado: julio de 2010.

[29] OGRE FORUMS. http://www.ogre3d.org/forums/. Consultado: enero de 2010.

[30] BLODDYMESS. http://www.ogre3d.org/tikiwiki/Spacegaier+Tutorials&structure=Libraries. Consultado: julio de 2010.

[31] BLENDER. http://www.blender.org/. Consultado: julio de 2010.

[32] FLOUR. http://www.ogre3d.org/tikiwiki/Flour. Consultado: julio de 2010.

[33] OGREHAPTICS. Documento pdf en línea disponible en ogrehaptics.sourceforge.net/manual/manual.pdf. Consultado: enero de 2010.

Anexo A. Instalación de los Componentes Software de la Aplicación

A continuación se describen los pasos para la instalación de los programas utilizados para la realización de la aplicación: Ogre 3D, OgreHaptics, PhysX y NxOgre. Así como los drivers necesarios para que funcione correctamente el dispositivo háptico.

A.1. Instalación de Drivers de Novint Falcon

Para empezar a trabajar con la interfaz háptica Novint Falcon es necesario leer el manual de usuario de la interfaz "User Manual" http://home.novint.com/download/User%20Manual.pdf y después instalar los drivers de la misma de acuerdo a la guía, estos se pueden encontrar en el siguiente enlace http://home.novint.com/support/download.php

1. Asegúrese de no conectar la interfaz hasta no haber instalado los drivers de forma adecuada.

2. Diríjase a la página 10 del manual que descargó y siga los pasos atentamente para instalar los drivers de manera correcta.

3. Una vez instalados los drivers, conecte la interfaz al computador. Dentro de la carpeta de los drivers encontrará un archivo de nombre "FalconTest" ejecútelo y pruebe el correcto funcionamiento de los leds y motores de la interfaz. Debe aparecer una ventana como la que se muestra en la Figura A.1

4. Después en el mismo enlace donde encontró los drivers puede descargar los juegos que trae la interfaz para que pueda probarla.

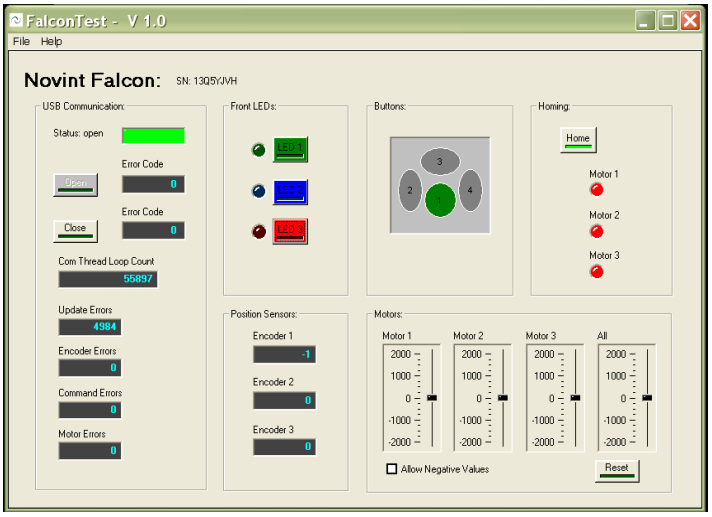

Figura A.1 Falcon Test.

NOVINT HDAL SDK

1. Para descargar este kid para desarrollo de software (SDK) se debe ingresar a la siguiente dirección http://home.novint.com/products/sdk.php, en la cual tendrá que digitar su dirección de correo electrónico al cual le llegará un enlace desde el cual podrá descargar el instalador.

2. Instalar el kit para desarrollo de aplicaciones software, para esto se debe ejecutar 2008-08-20_Novint_HDAL_SDK_2.1.3_setup.

3. Durante la instalación fijarse donde se crea la carpeta en la cual quedaran almacenados los archivos del SDK. Para este caso se creó en C:\Program Files (x86)\novint\HDAL_SDK_2.1.3

4. Una vez esté lista la instalación se debe chequear dos variables de entorno, para revisar esto debe dirigirse al panel de control-sistema-configuración de acceso remoto-opciones avanzadas-variables de entorno

   - NOVINT_DEVICE_SUPORT. Debe apuntar a la carpeta donde tú instalaste el SDK.

   - PATH. Debería comenzar con XXXX\bin, donde XXXX tiene la misma dirección que tiene NOVINT_DEVICE_SUPORT. Note el punto “.”; este te permite probar versiones de DLL en el folder de aplicación para probar sin disturbios el uso normal de las DLLs para cualquier otra aplicación.

5. Dentro de estos archivos se encuentran unos ejemplos que pueden probarse para ver su funcionamiento. Para ello es necesario abrir Microsoft Visual Studio 2005 Profesional Edition, luego Archivo-Abrir-Proyecto o solución, en la ventana que se despliega se debe buscar la carpeta donde

quedo guardado HDAL_SDK_2.1.3. , dentro de esta carpeta examples-Basic-vs2005-basic_opengl se selecciona el proyecto que aparece en este directorio basic_opengl y se da clic en abrir.

6. Antes de correr la aplicación se deben incluir los directorios necesarios, para esto en visual studio 2005 se debe ir a Herramientas-Opciones-Proyectos y soluciones-Directorios de VC++ en la pestaña que dice mostrar directorios se debe seleccionar la opción archivos de inclusión, dar clic en nueva línea (icono en forma de carpeta que aparece en a lado derecho de la ventana), dar clic en los "…" y buscar la dirección donde se encuentra HDAL\include. Ahora seleccione la opción archivos de biblioteca y repita el procedimiento anterior, pero ahora busque el directorio en el que se encuentra la carpeta HDAL\lib.

7. Ahora diríjase en visual studio 2005, a proyecto-propiedades de basic_opengl seleccione C\C++ luego General-Directorios de inclusión adicionales y de clic en "…" e incluya $(NOVINT_DEVICE_SUPORT)\include. Luego vaya a Vinculador-General-Directorios de bibliotecas adicionales, de clic en "…" y adicione $(NOVINT_DEVICE_SUPORT)\lib.

8. Compile el proyecto en modo Debug y busque el ejecutable en C:\Program Files (x86)\novint\HDAL_SDK_2.1.3\bin\Debug. Antes de correr el ejecutable debe copiar en este directorio en archivo glut.dll el cual se encuentra en C:\Program Files (x86)\novint\HDAL_SDK_2.1.3\examples\Basic\src.

A.2. Instalación de Ogre 3D

El ejecutable de instalación se puede descargar de http://sourceforge.net/projects/ogre/files/ogre/1.6.4/OgreSDKSetup1.6.4_VC80.exe/download. Se debe ejecutar el instalador y dejar las opciones por establecidas por defecto.

Para que Ogre pueda trabajar con Visual Studio es necesario instalar Ogrewizard. Este se puede descargar de la siguiente dirección:
http://sourceforge.net/projects/ogreconglo/files/ogresdkwizard80_Eihort_v1_4_2.zip/download

Ahora se debe editar el archivo OgreSDKAppWizard80.vsz y se debe cambiar "ABSOLUTE_PATH" por el directorio donde se ha descomprimido el Ogrewizard.
Después se deben copiar los tres archivos "OgreSDKAppWizard80.*" al directorio [Visual Studio]/VC/vcprojects.

A.3. Instalación de OgreHaptics

1. Para usar este kid de desarrollo de software de deben descargar el instalador de Ogre SDK 1.6.4, en cual se puede encontrar en este directorio http://sourceforge.net/projects/ogre/files/ogre/1.6.4/OgreSDKSetup1.6.4_VC80.exe/download. Se debe ejecutar el instalador y dejar las opciones por establecidas por defecto.

2. Luego se debe descargar OgreHaptics SDK V2.0, el cual se puede conseguir aquí http://sourceforge.net/projects/ogrehaptics/files/ogrehaptics/OgreHapticsSDKSetup0.2_vc80.exe/ . Después de descargarlo se debe ejecutar el instalador.

3. Se deben descargar unas bibliotecas de boost como threading library, las cuales se pueden descargar aquí http://sourceforge.net/projects/boost/files/boost/1.40.0/boost_1_40_0.zip/download. Después de descargar y descomprimir los archivos se debe ejecutar el archivo booststrap.bat, el cual generará un ejecutable llamado bjam.exe, deberá correr este archivo, el cual generara las bibliotecas.

4. Cuando vaya a ejecutar alguna aplicación no olvide incluir todos los directorios que requiera, esto lo puede hacer dentro de Visual Studio 2005 en Herramientas-Opciones-Proyectos y soluciones-Directorios de VC++, aquí puede incluir los directorios de archivos ejecutables, de inclusión, de referencia, de biblioteca y de código fuente. Debe incluir en estos directorios la dirección de las librerías de boost que acabo de instalar, debe incluir en los archivos de biblioteca la dirección boost-1.40.0\boost-1.40.0\stage\lib, esta se encuentra donde usted descomprimió los archivos.

5. Antes de probar los demos debe tener en cuenta que este kit no viene prediseñado para trabajar con el dispositivo Novint Falcon, por ello debe cambiar algunos parámetros dentro de los archivos, por ejemplo debe activar el plugin para Force FX y activar el de Novint Falcon en el archivo HapticsPlugins.cfg, el cual se encuentra en C:\OgreHapticsSDK\bin\release.

6. Ahora puede probar los demos que se encuentran en C:\OgreHapticsSDK\demos\scripts, se recomienda probar el demo Demo_Weapons_vc8.vcproj y compilarlo en modo Release desde Visual Studio 2005.

7. Cuando se trabaja en una aplicación propia no se debe olvidar incluir en los directorios la dirección en la que se encuentran las figuras y archivos propios de la aplicación diseñada. Se deben copiar también los archivos OgreHapticsMain.dll, hapticsresources.cfg, HapticsPlugins.cfg y HapticsRenderSystem_Falcon.dll que se encuentran en C:\OgreHapticsSDK\bin\release a este directorio C:\OgreSDK\bin\release.

A.4. Instalación del Motor Físico PhysX

1. Para poder instalar el motor físico es necesario registrarse en la página de Ageia http://devsupport.ageia.com y dar click en registrar. En el registro se le pedirá una dirección de correo electrónico, asegúrese que esta dirección sea válida pues será necesario para poder terminar el registro y descargar el software de manera correcta.

2. Una vez el registro este hecho, se le permitirá descargar el software de Ageia, el primer software para descargar es PhysX_9.10.0513_SystemSoftware.exe

PhysX requiere que el computador donde se va a trabajar tenga instalado el software del sistema antes de instalar el SDK. Para obtener el software del sistema ingrese a http://devsupport.ageia.com, en esta página haga clic en la pestaña "Descargas" en la página de inicio (alrededor de la mitad de la página a la derecha). Haga clic para instalar PhysX 9.10.0513 SystemSoftware.exe. Esto le llevará a través de un asistente de instalación sencilla, de nuevo sólo tiene que utilizar los valores por defecto.

3. En el menú lateral a la izquierda de la página de descargas, haga clic en el pequeño "+" al lado de Descargas para abrir el menú. Se mostrará una lista de todas las versiones del SDK de PhysX. La última es la versión SDK PhysX_2.8.3 así que haga clic ahí. El primer enlace que dice "PhysX_2.8.3.21_for_PC_Core.exe (REQUIERE Software del sistema 9.10.05133)". Como acaba de instalar el software del sistema 9.10.05133 podemos seguir adelante y haga clic para instalar el SDK de PhysX. Al igual que con las demás instalaciones, sólo tiene que utilizar los valores por defecto no cambia nada.

4. Una vez instalados el software del sistema y el SDK se puede empezar a explorar los demos que trae PhysX. Los programas de ejemplos vienen con archivos de proyectos para Microsoft Visual Studio 8 y 9. Para poder correr estor archivos será necesario acceder a las DLL (Dynamic Link Library) del SDK, las cuales están localizadas en el directorio /bin/win32 (este incluye PhysXLoader.dll and NxCharacter.dll).

5. Cuando vaya a ejecutar alguna aplicación no olvide incluir todos los directorios que requiera, esto lo puede hacer dentro de Visual Studio 2005 en Herramientas-Opciones-Proyectos y soluciones-Directorios de VC++, aquí puede incluir los directorios de archivos de inclusión 'SDKs\Foundation\include', 'SDKs\Physics\include', 'SDKs\PhysXLoader\include', '*SDKs**Cooking**include*' y '*SDKs**NxCharacter**include*' de referencia, de biblioteca '*SDKs**lib**win32*\' y de código fuente.

6. Al crear un archivo ejecutable se debe asegurar que las DLL estén disponibles en la misma carpeta del ejecutable para su correcto funcionamiento.

A.5. Instalación de NxOgre

1. Para empezar la instalación diríjase a la siguiente dirección electrónica y descargue la versión de NxOgre BloodyMess 1.5.5
http://static.nxogre.org/releases/bloodymess/NxOgre.1.5.5.BloodyMess.zip

2. Después de instalar versión indicada de NxOgre es necesario ajustar las variables de entorno de PhysX y de NxOgre. Para ajustar la de PhysX vaya en Windows a Inicio-Panel de Control-Sistema-Propiedades Avanzadas del Sistema en la pestaña Avanzadas haga click en el botón de Variables de Entorno.

Desplácese por la lista de variables de usuario y asegúrese de que este OGRE_HOME con el valor de c: \ OgreSDK. La variable OGRE_HOME debería haber sido creada durante la instalación de Ogre. A continuación haga clic en "Nuevo ..." para agregar una nueva variable del sistema. En el diálogo variable que aparece escriba el nombre de la variable 'PHYSX_DIR' y el valor " C:\Program Files (x86)\NVIDIA Corporation\NVIDIA PhysX SDK\v2.8.3" este paso es MUY IMPORTANTE y de la misma manera para NxOGRE_DIR y el valor “C:\ NxOgre_Bloody”.

4. Ahora diríjase al directorio C:\NxOgre_Bloody\build\msvc y corra el proyecto NxOgre.VC8.vcproj en modo Release para poder generar las bibliotecas y dll necesarias para poder utilizar NxOgre. Haga lo mismo para los demás proyectos que se encuentren en este directorio como OgreRenderSystem.VC8.vcproj.

5. Cuando vaya a ejecutar alguna aplicación no olvide incluir todos los directorios que requiera, esto lo puede hacer dentro de Visual Studio 2005 en Herramientas-Opciones-Proyectos y soluciones-Directorios de VC++, aquí puede incluir los directorios de archivos de inclusión ' C:\NxOgre_Bloody\sdk’, 'C:\NxOgre_Bloody\sdk\bml', C:\NxOgre_Bloody\rendersystems\ogre', de referencia, de biblioteca C:\NxOgre_Bloody\sdk'.

Por último copie NxOgre.dll a 'C:\OgreSDK\bin\release\' y asegúrese que las dll de PhysX se encuentren en ese directorio para que el ejecutable corra sin problemas.

A.6. Instalación de Flour

Flour es una pequeña herramienta que permite la conversión de diferentes tipos de formatos para poder trabajar en el mundo de NxOgre. Como Ogre Mesh a mallas de triángulos, esqueletos o cuerpos convexos.

1. Generación de un archivo .nxs en Flour

En primer lugar, tenemos que generar un archivo con los datos de malla física del Ogre Mesh para la aplicación. Para hacer la conversión, en la misma carpeta

donde se encuentra Flour.exe se incluye el Ogre Mesh a convertir (que sea un cubo simple, porque esto realmente no tendría ningún sentido en absoluto).

Ahora se puede ejecutar Flour a través de la línea de comandos, si esta en Windows XP vaya Run-cmd y cambie el directorio hasta que se encuentre en la carpeta del Flour.exe. En Microsoft Windows Vista, puede pulsar Mayús y hacer clic a la izquierda en la carpeta que se encuentra en Flour.exe y seleccione Abrir ventana de comandos aquí para abrir automáticamente la línea de comandos, y ya estará establecido en esta carpeta.

Una vez hecho esto hay que decirle a Flour lo que debe hacer usando la siguiente sintaxis:

flour convert in:infile, into:type, out:outfile

En este caso se necesita convertir un ogre mesh a un archivo .nxs ogre de tipo triangle mesh. Entonces la conversión debe escribirse asi:

flour convert in:Abdomen.mesh, into:triangle, out:Abdomen.nxs

Si la conversión fue realizada de manera correcta entonces Flour debe entregar un Ok en la pantalla y puede ser usado en BloodyMess. En el campo into se debe poner el tipo de archivo al cual se quiere convertir puede ser de tipo triangle, convex, skeleton y heightFields.

Limites para tener en cuenta:

Para un Convex Mesh (*.nxs) debe tener un límite de vértices de 256.
Para Triangle Meshes (*.nxs) no se pueden convertir OgreMesh que usen vértices compartidos.
Skeleton meshes (*.xsk) tiene un límite de vértices de 64 and y un límite de bordes de 256.
HeightFields (*.xhf) pueden ser convertidos desde 8-bits o 16 bits de imágenes en escala de grises, sin la información del encabezado

Anexo B. Código Fuente de la Aplicación

En este anexo se muestra todo el código fuente que forma parte de la aplicación realizada.

Este código fuente consta de un programa principal donde se hace un llamado al archivo de cabecera llamado Escena.h.

En el archivo Escena.h se encuentra la clase principal llamada LapBotApplication que contiene la creación de la escena quirúrgica: luces, cámara, cuarto, mesa, abdomen, mundo en NxOgre y los cuerpos envolventes necesarios para la detección de colisiones.

```
#ifndef __Escena_h_
#define __Escena_h_

#include "Iteracion.h"
#include "ExampleApplication.h"
// Para la biblioteca NxOgre
#include <NxOgre.h>
#include <NxOgreOGRE3D.h>


class LapBotApplication : public OgreHapticsDemoApplication, public NxOgre::Callback
{
public:
	LapBotApplication()
		: OgreHapticsDemoApplication()
	{
	}

	~LapBotApplication()
	{
		if(mWindow)
		{
		delete mWindow;
		}
	}

protected:
	NxOgre::World* mWorld;
	NxOgre::Scene* mScene;
	NxOgre::TimeController* mTimeController;

	//Función que crea la escena.
	void createScene(void)
	{
		 // Se ajustan las luces del ambiente
      mSceneMgr->setAmbientLight(ColourValue(0.5f, 0.5f, 0.5f));
```

```
        // Crea una luz
        Light* l = mSceneMgr->createLight("MainLight");
        l->setPosition(20, 80, 50);

                        // Crea el mundo de NxOgre
        mWorld = NxOgre::World::createWorld();

        // Crea una descripcion de la escena para NxOgre
        NxOgre::SceneDescription sceneDesc;
        sceneDesc.mGravity = NxOgre::Vec3(0, 0, 0);//-9.8f
        sceneDesc.mName = "DemoScene";

        // Crea la escena de NxOgre
        mScene = mWorld->createScene(sceneDesc);

        // Se ajustan algunos valores fisicos para la escena
        mScene->getMaterial(0)->setStaticFriction(0.5);
        mScene->getMaterial(0)->setDynamicFriction(0.5);
        mScene->getMaterial(0)->setRestitution(0.1);

        // Crea el sistema de renderizado
        mRenderSystem = new OGRE3DRenderSystem(mScene);

        //Crea el controlador de tiempo para NxOgre
        mTimeController = NxOgre::TimeController::getSingleton();


    float escuerpo=0.1, esroom=8;          //constantes de escalamiento

            //AMBIENTE
SceneNode *nodoRoom = mSceneMgr->getRootSceneNode()->createChildSceneNode();
nodoRoom->scale(esroom, esroom, esroom);

                        // Habitacion
// Generando el piso
MaterialPtr mpiso = MaterialManager::getSingleton().create("Mpiso",
ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);
TextureUnitState* tpiso = mpiso->getTechnique(0)->getPass(0)->createTextureUnitState("wood_15.jpg");
tpiso->setTextureScale(0.05,1);

Entity *piso = mSceneMgr->createEntity("piso",mSceneMgr->PT_CUBE);
piso ->setMaterialName("Mpiso");
SceneNode *npiso = nodoRoom->createChildSceneNode(Vector3(1,-3.35,0));
npiso->attachObject(piso);
npiso->scale(0.2, 0.0015, 0.2);

// Generando las paredes
MaterialPtr mpared = MaterialManager::getSingleton().create("mpared",
ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);
TextureUnitState* tpared = mpared->getTechnique(0)->getPass(0)->createTextureUnitState("marble_7.jpg");

Entity *pared = mSceneMgr->createEntity("pared",mSceneMgr->PT_CUBE);
pared->setMaterialName("mpared");
SceneNode *npared = npiso->createChildSceneNode(Vector3(50,4000,0));
npared->attachObject(pared);
npared->roll(Degree(90));
npared->scale(0.6,1,1);

Entity *pared2 = pared->clone("pared2");
SceneNode *npared2 = npiso->createChildSceneNode(Vector3(-50,4000,0));
npared2->attachObject(pared2);
npared2->roll(Degree(90));
npared2->scale(0.6,1,1);

Entity *pared3 = pared->clone("pared3");
SceneNode *npared3 = npiso->createChildSceneNode(Vector3(0,4000,-50));
npared3->attachObject(pared3);
npared3->pitch(Degree(90));
npared3->scale(1,1,0.6);
```

```
Entity *pared4 = pared->clone("pared4");
SceneNode *npared4 = npiso->createChildSceneNode(Vector3(0,4000,50));
npared4->attachObject(pared4);
npared4->pitch(Degree(90));
npared4->scale(1,1,0.6);

// Generando el techo
Entity *techo = pared->clone("techo");
SceneNode *ntecho = npiso->createChildSceneNode(Vector3(0,8000,0));
ntecho->attachObject(techo);

// Generando luces
mSceneMgr->setAmbientLight(ColourValue(0, 0, 0));
mSceneMgr->setShadowTechnique(SHADOWTYPE_NONE); //sin sombras

                        float df=45;                        // Distancia entre los focos y el centro del techo
Entity *Foco1         = mSceneMgr->createEntity("Foco1", "sphere.mesh");
Light  *l1  = mSceneMgr->createLight("Luz1");
l1->setPosition(df,-50,df); //respecto al techo
SceneNode *nFoco1 = ntecho->createChildSceneNode(Vector3(df,-100,df));
nFoco1->attachObject(Foco1);
nFoco1->scale(0.01,1,0.01);
nFoco1->attachObject(l1);

Entity *Foco3         =  Foco1->clone("Foco3");
Light  *l3  = mSceneMgr->createLight("Luz3");
l3->setPosition(-df,-50,df); //respecto al techo
SceneNode *nFoco3 = ntecho->createChildSceneNode(Vector3(-df,-100,df));
nFoco3->attachObject(Foco3);
nFoco3->scale(0.01,1,0.01);
nFoco3->attachObject(l3);

Entity *Foco4         = Foco1->clone("Foco4");
Light  *l4  = mSceneMgr->createLight("Luz4");
l4->setPosition(0,-50,-df); //respecto al techo
SceneNode *nFoco4 = ntecho->createChildSceneNode(Vector3(0,-100,-df));
nFoco4->attachObject(Foco4);
nFoco4->scale(0.01,1,0.01);
nFoco4->attachObject(l4);

// Generando la mesa y el abdomen
MaterialPtr matmesa = MaterialManager::getSingleton().create("MatMesa",
ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);
TextureUnitState* tmesa = matmesa->getTechnique(0)->getPass(0)->createTextureUnitState("Water01.jpg");

MaterialPtr mat3 = MaterialManager::getSingleton().create("MatAbdomen",
ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);
Technique *tec = mat3->createTechnique();
tec->createPass();
TextureUnitState* t3 = mat3->getTechnique(0)->getPass(0)->createTextureUnitState("piel.jpg");
t3->setColourOperation(LBO_MODULATE );
mat3->getTechnique(0)->getPass(0)->setSceneBlending( SBT_TRANSPARENT_COLOUR);

Entity *Mesa             = mSceneMgr->createEntity("Mesa", "mesa.mesh");
Mesa ->setMaterialName("MatMesa");
SceneNode *nodoMesa  = mSceneMgr->getRootSceneNode()->createChildSceneNode(Vector3( -1.325, -2.65, 0));
nodoMesa->attachObject(Mesa);
nodoMesa->pitch( Degree(-90));

//Abdomen
// Carga el Mesh triangulizado para trabajar con el Abdomen
NxOgre::ResourceSystem::getSingleton()->openArchive("media", "file:C:/OgreSDK/media");
      NxOgre::Mesh* triangleMesh = NxOgre::MeshManager::getSingleton()->load("media:Mesh.nxs");

      // Crea TriangleGeometry para el Abdomen
      NxOgre::TriangleGeometry* triangleGeometry = new NxOgre::TriangleGeometry(triangleMesh);

      // Crea el cuerpo envolvente para la pinza
Body = mRenderSystem->createBody(new NxOgre::Sphere(0.18),NxOgre::Vec3(10.475, 2, 8.8), "pinza.mesh");//
 delete(Body->getEntity());
```

```
 // Para el Abdomen
Entity *Abdomen = mSceneMgr->createEntity("Abdomen", "Mesh.mesh");
Abdomen ->setMaterialName("MatAbdomen");
SceneNode *nodoAbdomen =  mSceneMgr->getRootSceneNode()->createChildSceneNode(Vector3( 10.475, -2.65, 8.8));
nodoAbdomen->attachObject(Abdomen);


// Crea el Volumen para la deteccion de Colisiones
mVolume =  mScene->createVolume(triangleGeometry, NxOgre::Matrix44(NxOgre::Vec3(10.475, -2.65, 8.8)), this,
NxOgre::Enums::VolumeCollisionType_All);

// Generando los Robots
CrearLapBot2(mSceneMgr,nodoMesa, Vector3( 25.4, -17.2, -1.09)); //x = ancho aprox mesa
//y = distancia del centro mesa a la ubicacion del robot
//z = eje base colineal con la mesa
CrearLapBot1(mSceneMgr,nodoMesa, Vector3( -1.786, -17.2, -1.09)); //x = mitad de ancho de la base
//y = distancia del centro mesa a la ubicacion del robot
//z = eje base colineal con la mesa


  // LISTO EL AMBIENTE
  // Posiciona la Camara
        mCamera->setPosition (10, 1, 73);
        mCamera->lookAt(Vector3 (10, 0, 0));
        mCamera->setNearClipDistance (0.1);
  }

        // Llamado al FrameListener
        void createFrameListener(void)
        {
                LapBotListener* lapBotListener = new LapBotListener(mWindow, mCamera, mSystem, mDevice);
                mFrameListener = lapBotListener;
                mFrameListener->showDebugOverlay(true);
                mRoot->addFrameListener(mFrameListener);
        }
// Función para la deteccion de colisiones
void onVolumeEvent(NxOgre::Volume* volume, NxOgre::Shape* volumeShape, NxOgre::RigidBody* rigidBody,
NxOgre::Shape* rigidBodyShape, unsigned int collisionEvent)
  {
    if(collisionEvent == NxOgre::Enums::VolumeCollisionType_OnEnter)
    {
        collision = true;
    }
  }
};
#endif
```

En el archivo Robot.h se encuentra la creación del robot para cirugía laparoscópica LapBot.

```
//Robot.h

#include "OgreHapticsDemoApplication.h"
#include "OgreHapticsDemoFrameListener.h"
#include "OgreHaptics.h"

#ifndef __Robot_h_
#define __Robot_h_

// Variables para el robot 1
SceneNode *nodor1;
SceneNode *nodoBrazo;
SceneNode *nodoAntebrazo;
SceneNode *nodot4;
SceneNode *nodot5;
SceneNode *nodot6;
SceneNode *nodot7;
SceneNode *nodot8;
```

```
SceneNode *nodot9;
SceneNode *nodopinza, *nodogancho;
SceneNode *nodolapiz;
ManualObject *lapiz;
Entity *pinza, *gancho;

// Variables para el robot 2
SceneNode *nodor12;
SceneNode *nodoBrazo2;
SceneNode *nodoAntebrazo2;
SceneNode *nodot42;
SceneNode *nodot52;
SceneNode *nodot62;
SceneNode *nodot72;
SceneNode *nodot82;
SceneNode *nodot92;
SceneNode *nodopinza2;
SceneNode *nodolapiz2;
ManualObject *lapiz2;

void CrearLapBot1(SceneManager* mSceneMgr, SceneNode *nodoPadre, const Vector3 &posicion)
{
// Creando el robot 1
// Las dimensiones y angulos se obtienen de las tablas del robot.
// la relacion dimensiones en mm entre los planos y pixeles de pantalla es 25.409

// Crea materiales
MaterialPtr mat1=
MaterialManager::getSingleton().create("MatBase",ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);
TextureUnitState* t1 = mat1->getTechnique(0)->getPass(0)->createTextureUnitState("NMHollyBumps.png");
MaterialPtr materialapiz = MaterialManager::getSingleton().create("materialapiz","debugger");
materialapiz->setReceiveShadows(false);
materialapiz->getTechnique(0)->setLightingEnabled(true);
materialapiz->getTechnique(0)->getPass(0)->setDiffuse(0,0,1,0);
materialapiz->getTechnique(0)->getPass(0)->setAmbient(0,0,1);
materialapiz->getTechnique(0)->getPass(0)->setSelfIllumination(0,0,1);

//Base
Entity *Base              = mSceneMgr->createEntity("Base", "Base.mesh");
Base ->setMaterialName("MatBase");
SceneNode *nodoBase = nodoPadre->createChildSceneNode(posicion);
nodoBase->attachObject(Base);
nodoBase->roll(Degree(180));

//Ejes
Entity *ejeBase = mSceneMgr->createEntity("ejeBase", "axes.mesh");
SceneNode *nodoejeBase = nodoBase->createChildSceneNode(Vector3( 0, 0, 0));
nodoejeBase->attachObject(ejeBase);
nodoejeBase->roll ( Degree(180));

//Nodo de soporte para r1
SceneNode *r1s = nodoejeBase->createChildSceneNode(Vector3( 0, 0, 0));
r1s->translate(Vector3(0,0,-15.74)); // la relacion de dimensiones es 25.409

//r1
Entity *r1 = mSceneMgr->createEntity("r1", "r1.mesh");
r1 ->setMaterialName("MatBase");  //MatExterno
nodor1 = r1s->createChildSceneNode(Vector3( 0, 0, 0));
nodor1->attachObject(r1);

//brazo
Entity *Brazo = mSceneMgr->createEntity("Brazo", "Brazo.mesh");
Brazo ->setMaterialName("MatBase");
nodoBrazo = nodor1->createChildSceneNode(Vector3( 0,0,17.12));
nodoBrazo->attachObject(Brazo);

//Antebrazo
Entity *Antebrazo = mSceneMgr->createEntity("Antebrazo", "Antebrazo.mesh");
Antebrazo->setMaterialName("MatBase");
nodoAntebrazo = nodoBrazo->createChildSceneNode(Vector3( 9.84, 0, -0.3));
```

```
nodoAntebrazo->attachObject(Antebrazo);

//t4
Entity *t4 = mSceneMgr->createEntity("t4", "t4.mesh");
nodot4 = nodoAntebrazo->createChildSceneNode(Vector3( 7.87, 0, 0));
nodot4->pitch( Degree(90));
nodot4->attachObject(t4);

//Nodo de soporte para t5
SceneNode *t5s = nodot4->createChildSceneNode(Vector3( 0, -3.7, 0));
t5s->pitch( Degree(90));
//t5
Entity *t5 = mSceneMgr->createEntity("t5", "t5.mesh");
t5->setMaterialName("MatBase");
nodot5 = t5s->createChildSceneNode(Vector3( 0, 0, 0));
nodot5->attachObject(t5);

//Nodo de soporte para t6
SceneNode *t6s = nodot5->createChildSceneNode(Vector3( 0, 0, 4.195));
t6s->pitch( Degree(-90));

//t6
Entity *t6 = mSceneMgr->createEntity("t6", "t6.mesh");
nodot6 = t6s->createChildSceneNode(Vector3( 0, 0, 0));
nodot6->attachObject(t6);

//Nodo de soporte para t7
SceneNode *t7s = nodot6->createChildSceneNode(Vector3( 0, -11.87, 0));
t7s->pitch( Degree(90));

//t7
Entity *t7 = mSceneMgr->createEntity("t7", "t7.mesh");
nodot7 = t7s->createChildSceneNode(Vector3( 0, 0, 0));
nodot7->attachObject(t7);

//Nodo de soporte para t8
SceneNode *t8s = nodot7->createChildSceneNode(Vector3( 0, 0, -0.057));
t8s->pitch( Degree(-90));
t8s->roll( Degree(-90));

//t8
Entity *t8 = mSceneMgr->createEntity("t8", "t8.mesh");
nodot8 = t8s->createChildSceneNode(Vector3( 0, 0, 0));
nodot8->attachObject(t8);

//Nodo de soporte para t9
SceneNode *t9s = nodot8->createChildSceneNode(Vector3( 0, 0, 0));
t9s->pitch( Degree(90));
t9s->yaw( Degree(90));

//t9
Entity *t9 = mSceneMgr->createEntity("t9", "t9.mesh");
nodot9 = t9s->createChildSceneNode(Vector3( 0, 0, 0));
nodot9->attachObject(t9);

//Pinza
pinza = mSceneMgr->createEntity("pinza", "pinza.mesh");
nodopinza = nodot9->createChildSceneNode(Vector3( 0, 0, 0.4));
nodopinza->attachObject(pinza);

//Gancho
/*gancho = mSceneMgr->createEntity("gancho", "gancho.mesh");
nodogancho = nodot9->createChildSceneNode(Vector3( 0, 0, 0.3936));
nodogancho->attachObject(gancho);
gancho->setVisible(false);*/

// Creando el lapiz
lapiz =  mSceneMgr->createManualObject("lapiz");
nodolapiz = nodoejeBase->createChildSceneNode();
nodolapiz->attachObject(lapiz);
```

```
}

void CrearLapBot2(SceneManager* mSceneMgr, SceneNode *nodoPadre, const Vector3 &posicion)
{
// Creando el robot2
// Las dimensiones y angulos se obtienen de las tablas del robot.
// la relacion dimensiones en mm entre los planos y pixeles de pantalla es 25.409

// Crea materiales
MaterialPtr mat1 = MaterialManager::getSingleton().create("MatBase",
ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);
TextureUnitState* t1 = mat1->getTechnique(0)->getPass(0)->createTextureUnitState("NMHollyBumps.png");

//Base
Entity *Base2 = mSceneMgr->createEntity("Base2", "Base.mesh");
Base2 ->setMaterialName("MatBase");
SceneNode *nodoBase2 = nodoPadre->createChildSceneNode(posicion);
nodoBase2->attachObject(Base2);

//Ejes
Entity *ejeBase2      = mSceneMgr->createEntity("ejeBase2", "axes.mesh");
SceneNode *nodoejeBase2 = nodoBase2->createChildSceneNode(Vector3( 0, 0, 0));
nodoejeBase2->attachObject(ejeBase2);
nodoejeBase2->roll ( Degree(180));

//Nodo de soporte para r1
SceneNode *r1s2 = nodoejeBase2->createChildSceneNode(Vector3( 0, 0, 0));
r1s2->translate(Vector3(0,0,0)); //(0,0,-15.74)// la relacion de dimensiones es 25.409

//r1
Entity *r12 = mSceneMgr->createEntity("r12", "r1.mesh");
r12 ->setMaterialName("MatBase");
nodor12 = r1s2->createChildSceneNode(Vector3( 0, 0, 0));
nodor12->attachObject(r12);

//brazo
Entity *Brazo2 = mSceneMgr->createEntity("Brazo2", "Brazo.mesh");
Brazo2 ->setMaterialName("MatBase");
nodoBrazo2 = nodor12->createChildSceneNode(Vector3(0,0,17.12));
nodoBrazo2->attachObject(Brazo2);
nodoBrazo2->setOrientation(Quaternion(Radian(-3.14159265/2),Vector3( 0,0,1)));

//Antebrazo
Entity *Antebrazo2 = mSceneMgr->createEntity("Antebrazo2", "Antebrazo.mesh");
Antebrazo2->setMaterialName("MatBase");
nodoAntebrazo2 = nodoBrazo2->createChildSceneNode(Vector3( 9.84, 0, -0.3));
nodoAntebrazo2->attachObject(Antebrazo2);

//t4
Entity *t42 = mSceneMgr->createEntity("t42", "t4.mesh");
nodot42 = nodoAntebrazo2->createChildSceneNode(Vector3( 7.87, 0, 0));
nodot42->pitch( Degree(90));
nodot42->attachObject(t42);

//Nodo de soporte para t5
SceneNode *t5s2 = nodot42->createChildSceneNode(Vector3( 0, -3.7, 0));
t5s2->pitch( Degree(90));

//t5
Entity *t52 = mSceneMgr->createEntity("t52", "t5.mesh");
t52->setMaterialName("MatBase");
nodot52 = t5s2->createChildSceneNode(Vector3( 0, 0, 0));
nodot52->attachObject(t52);

//Nodo de soporte para t6
SceneNode *t6s2 = nodot52->createChildSceneNode(Vector3( 0, 0, 4.195));
t6s2->pitch( Degree(-90));

//t6
Entity *t62 = mSceneMgr->createEntity("t62", "t6.mesh");
```

```
nodot62 = t6s2->createChildSceneNode(Vector3( 0, 0, 0));
nodot62->attachObject(t62);

//Nodo de soporte para t7
SceneNode *t7s2 = nodot62->createChildSceneNode(Vector3( 0, -11.87, 0));
t7s2->pitch( Degree(90));

//t7
Entity *t72 = mSceneMgr->createEntity("t72", "t7.mesh");
nodot72 = t7s2->createChildSceneNode(Vector3( 0, 0, 0));
nodot72->attachObject(t72);

//Nodo de soporte para t8
SceneNode *t8s2 = nodot72->createChildSceneNode(Vector3( 0, 0, -0.057));
t8s2->pitch( Degree(-90));
t8s2->roll( Degree(-90));

//t8
Entity *t82 = mSceneMgr->createEntity("t82", "t8.mesh");
nodot82 = t8s2->createChildSceneNode(Vector3( 0, 0, 0));
nodot82->attachObject(t82);

//Nodo de soporte para t9
SceneNode *t9s2 = nodot82->createChildSceneNode(Vector3( 0, 0, 0));
t9s2->pitch( Degree(90));
t9s2->yaw( Degree(90));

//t9
Entity *t92 = mSceneMgr->createEntity("t92", "t9.mesh");
nodot92 = t9s2->createChildSceneNode(Vector3( 0, 0, 0));
nodot92->attachObject(t92);

//Pinza
Entity *pinza2 = mSceneMgr->createEntity("pinza2", "pinza.mesh");
nodopinza2 = nodot92->createChildSceneNode(Vector3( 0, 0, 0.3936));
nodopinza2->attachObject(pinza2);

// Creando el lapiz
lapiz2 =  mSceneMgr->createManualObject("lapiz2");
nodolapiz2 = nodoejeBase2->createChildSceneNode();
lapiz2->setDynamic(true);
nodolapiz2->attachObject(lapiz2);
}
#endif
```

En el archivo Iteración.h se encuentra una clase llamada EffectsListener que se ejecuta en cada frame de la aplicación, es decir esta es la clase que siempre se va a estar ejecutando una vez se haya creado la interfaz de usuario: la escena y el robot.

En esta clase se capturan los dispositivos de entrada como el ratón, el teclado y la interfaz háptica Novint Falcon. También dentro de esta clase se realiza el posicionamiento del robot y la realimentación de fuerza al dispositivo háptico en caso de detectarse una colisión.

```
#ifndef __Iteracion_h_
#define __Iteracion_h_
// Para OgreHaptics
#include "OgreHapticsDemoApplication.h"
#include "OgreHapticsDemoFrameListener.h"
#include "OgreHaptics.h"
// Para el teclado y el ratón
#include <OIS.h>
// Para Crear LapBot
#include "Robot.h"
 // Para la Deteccion de Colisiones
#include <NxOgre.h>
```

```
#include <NxOgreOGRE3D.h>
// Para la Realimentacion de Fuerza
#include <hdl/hdl.h>
#include <hdlu/hdlu.h>

// Variables de NxOgre
NxOgre::Vec3 pos;
OGRE3DBody* Body;
NxOgre::Volume* mVolume;
OGRE3DRenderSystem* mRenderSystem;

// Variables para la detección de colisiones y realimentación de fuerza
bool collision(false);
double force[3];
Vector3 pante;
Vector3 pactual;
int i=0,j=0;

class LapBotListener : public OgreHapticsDemoFrameListener

{
public:
	LapBotListener(RenderWindow* win, Camera* cam,
		OgreHaptics::System* system, OgreHaptics::Device* device)
		: OgreHapticsDemoFrameListener(win, cam, system, device),
		mShutdownRequested(false)
	{
		// t4 es estático
		nodot4->roll(Radian (3.14159265/4));

      mTimeController = NxOgre::TimeController::getSingleton();

		if (mDevice)
		{
			mDevice->setDeviceListener(this);
		}
	}

	  bool frameStarted(const FrameEvent& evt)
		{
			mTimeController->advance(evt.timeSinceLastFrame);
			return OgreHapticsDemoFrameListener::frameStarted(evt);
	   }
		void requestShutdown(void)
		{
			mShutdownRequested = true;
		}

// Variables para el MGI
double sx,sy,sz,nx,ny,nz,ax,ay,az,x1,y1,z1,p6,t2aux,t3aux,p8y;
double
vx,tx,ty,tz,t4,D1,D2,R1,R2,R3,x,y,z,K,M,N,r1,D,B,E1,E2,t2a1,t2a2,t2a3,c2,s2,t2,t3,s23,c23,c3,s3,c4,s4,A2,B2,t5,s5,c5,A1,B1
,t6,c6,s6,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,auxa,aur1,aur2,au3,au4,au5,au6,au7,au8,au9,au10,au11,t7,c8,s8,t8,t9;

// Esta funcion se repite cada frame
	bool frameRenderingQueued(const FrameEvent& evt)
	{
		mMouse->capture();
		mKeyboard->capture();
		if (mDevice)
		{
			mSystem->update();
			mDevice->capture();

// Para el posicionamiento de LapBot por medio del MGI

Vector3 devicePos = mDevice->getPosition(); //Lectura de la Posicion de la Interfaz Haptica
x1 = devicePos.x;
y1 = devicePos.y;
	z1 = -devicePos.z;
```

```
            x = 0.345+x1/400;
            y = 0.1952+z1/357;
            z = 0.2+y1/450;

            // Orientación
            sx = 1;
            sy = 0;
            sz = 0;
            nx = 0;
            ny = -1;
            nz = 0;
            ax = 0;
            ay = 0;
            az = -1;

            // Trocar
            tx=0.345;
            ty=0.075;
            tz=0.1952;

            // Valores fijos
            t4 = 3.14159265/4;
            D1 = 0.25;
            D2 = 0.2;
            R1 = 0.2;
            R2 = 0.3;
            R3 = 0.000001;

            // Calculo del MGI
K =  -R1*cos(t4)+az*R3-z;
M    =    (pow(az,2)+pow(ay,2)+pow(ax,2))*pow(R3,2)+(2*ty*ay-2*ay*y-2*ax*x-2*az*z+2*tx*ax+2*az*tz)*R3-2*z*tz+pow(y,2)-
2*tx*x-2*y*ty+pow(z,2)+pow(tx,2)+pow(tz,2)+pow(x,2)+pow(ty,2);
N = pow((az*R3-z+tz),2);

         //La solución para r1 es:
         if (z>tz)
            r1=-sqrt(((pow(R2,2))*N)/M) - K;
         else
            r1 = sqrt(((pow(R2,2))*N)/M) - K;

         D  = (-sin(t4)*R1-D2)*(-az*R3+z-tz);
         B  = -D1*(-az*R3+z-tz);
         E1 = -ty*(-az*R3+z-tz)+(tz+cos(t4)*R1-r1)*(-ay*R3+y-ty);
         E2 =  (tz+cos(t4)*R1-r1)*(-ax*R3+x-tx)-tx*(-az*R3+z-tz);

         // Las soluciones para t2 y t3 son:
         t2a1 = pow(E1,2) + pow(E2,2);
         t2a2 = (t2a1 + pow(B,2) - pow(D,2))/(2*B);
         t2a3 = sqrt(t2a1 - pow(t2a2,2));

         if (z>tz)
         {
            c2 = (E2*t2a2 - E1*t2a3)/t2a1;
            s2 = (E1*t2a2 + E2*t2a3)/t2a1;
         }
         else
         {
            c2 = (E2*t2a2 + E1*t2a3)/t2a1;
            s2 = (E1*t2a2 - E2*t2a3)/t2a1;
         }

         t2 = atan2(s2,c2);
         t3 = atan2(E1-B*s2, E2-B*c2) - t2;


         if (z>tz)
            t3=t3+3.14159265;

         s23 = sin(t2+t3);
```

```
c23 = cos(t2+t3);
c3  = cos(t3);
s3  = sin(t3);
s4  = sin(t4);
c4  = cos(t4);

p6=s23*s4*R1 + s23*D2 + s2*D1;   //para limitar el eje y de las articulaciones t2 y t3
if (p6>=(ty-0.01))
{
  t2=t2aux;
  t3=t3aux;
}


A2 = -r1*s4*ty + tz*s4*ty + D1*c4*tx*s3 + pow(ty,2)*c4*c2*s3 +pow(ty,2)*c4*s2*c3+pow(D2,2)*c2*s3*c4-
D1*c4*tx*s3*pow(c2,2)-r1*s2*c3*R1*pow(c4,2)+s2*D1*c4*D2-2*D2*s2*pow(c3,2)*c4*tx*c2-2*c4*R1*s4*pow(c2,2)*c3*tx*s3-
2*D2*pow(c2,2)*s3*c4*tx*c3+D2*c2*s3*c4*D1*c3+2*D2*pow(c2,2)*c4*ty*pow(c3,2)-
2*D2*c2*s3*c4*ty*s2*c3+D2*c2*s3*c4*s4*R1+s2*D1*c4*s4*R1*pow(c3,2)+D2*c3*c4*tx*s3+c4*R1*s4*s2*tx*c2+D2*c2*c4*tx*
s2+s2*pow(D1,2)*c4*c3+tz*s2*c3*R1*pow(c4,2)+tz*c2*s3*R1*pow(c4,2)+D2*s2*c3*c4*s4*R1+D2*s2*pow(c3,2)*c4*D1+r1*s
4*s2*c3*D2+ty*c4*D1*c3*pow(c2,2)+c4*R1*s4*s3*tx*c3-ty*c4*D2-2*ty*c4*D1*c3-D2*pow(c3,2)*c4*ty-tz*s4*s2*D1-
tz*c2*s3*R1-tz*s2*c3*R1-D2*pow(c2,2)*c4*ty+pow(D2,2)*s2*c3*c4+c4*R1*s4*c2*c3*D1*s3-s2*D1*c4*ty*c2*s3-
s2*D1*c4*tx*c2*c3-
2*c4*R1*s4*c2*pow(c3,2)*tx*s2+r1*c2*s3*R1+r1*s4*c2*s3*D2+ty*c4*tx*c2*c3+2*c4*R1*s4*pow(c2,2)*pow(c3,2)*ty-
ty*c4*tx*s2*s3-ty*c4*s4*R1*pow(c2,2)-r1*c2*s3*R1*pow(c4,2)-ty*c4*s4*R1*pow(c3,2)+r1*s2*c3*R1-tz*s4*c2*s3*D2-
tz*s4*s2*c3*D2-2*c4*R1*s4*c2*c3*ty*s2*s3+r1*s4*s2*D1;
B2=s2*D1*tx*c2*s3-D2*c2*D1-s2*pow(D1,2)*s3+2*R1*c2*s3*s4*tx*s2*c3+2*D2*s2*c3*tx*c2*s3-
2*R1*pow(c2,2)*s4*tx*pow(c3,2)-2*R1*pow(c2,2)*s3*s4*ty*c3-2*D2*s2*pow(c3,2)*ty*c2-2*D2*pow(c3,2)*tx*pow(c2,2)-
D1*tx*c3*pow(c2,2)-ty*tx*c2*s3-2*R1*s2*pow(c3,2)*s4*ty*c2-2*D2*pow(c2,2)*s3*ty*c3-ty*tx*s2*c3-R1*c2*s4*D1-
ty*D1*s3*pow(c2,2)+D2*c3*ty*s3-pow(ty,2)*s2*s3+D2*pow(c3,2)*tx+D2*pow(c2,2)*tx+2*ty*D1*s3-
s2*D1*ty*c2*c3+R1*pow(c3,2)*s4*tx+D2*c2*ty*s2+D2*c2*D1*pow(c3,2)+R1*pow(c2,2)*s4*tx+R1*c2*s4*D1*pow(c3,2)+R1*c
2*s4*ty*s2+R1*c3*s4*ty*s3-D2*s2*c3*D1*s3-R1*s2*c3*s4*D1*s3+D1*tx*c3+pow(ty,2)*c2*c3;

        //Las solución para t5:
        t5 = atan2(-B2,A2);

        s5 = sin(t5);
        c5 = cos(t5);

A1=-D1*c4*c5*s3+D2*s5+s4*R1*s5+tx*c4*c5*c2*s3+tx*s5*s2*s3+D1*s5*c3+tx*c4*c5*s2*c3-ty*c4*c5*c2*c3-
ty*s5*c2*s3+ty*c4*c5*s2*s3-ty*s5*s2*c3-tx*s5*c2*c3;
B1 = -D1*s4*s3+tx*s4*s2*c3+ty*s4*s2*s3-ty*s4*c2*c3+tx*s4*c2*s3;

        // Para t6:
        t6 = atan2(-B1,A1);

        s6 = sin(t6);
        c6 = cos(t6);

        // para garantizar que la pinza este denro del abdomen sobre el eje y

        p8y= -(-(s23*c4*c5-c23*s5)*s6-s23*s4*c6)*R2+s23*s4*R1+s23*D2+s2*D1;
        if(p8y<ty){
        t5=t5+3.14159265;
        //recalculando p6
        s5=sin(t5);
        c5=cos(t5);
A1=-D1*c4*c5*s3+D2*s5+s4*R1*s5+tx*c4*c5*c2*s3+tx*s5*s2*s3+D1*s5*c3+tx*c4*c5*s2*c3-ty*c4*c5*c2*c3-
ty*s5*c2*s3+ty*c4*c5*s2*s3-ty*s5*s2*c3-tx*s5*c2*c3;
B1=-D1*s4*s3+tx*s4*s2*c3+ty*s4*s2*s3-ty*s4*c2*c3+tx*s4*c2*s3;
        t6=atan2(-B1,A1);
        s6=sin(t6);
        c6=cos(t6);
        }

        a1  = s23*c5;
        a2  = c23*c5;
        a3  = s23*s5;
        a4  = c23*s5;
        a5  = s23*c4;
        a6  = c23*c4;
```

```
            a7  = s23*s4;
            a8  = c23*s4;
            a9  = s4*s5;
            a10 = c4*s6;

            auxa = s4*c6;
            aur1 = s4*c5*s6;
            aur2 = c4*c6;

            au3 = a8*s6;
            au4 = s5*s23*c6;
            au5 = c6*a6*c5;
            au6 = a7*s6;
            au7 = c6*a4;
            au8 = c6*a5*c5;
            au9 = s4*c5*c6;
            au10= s6*a6*c5;
            au11= s6*a5*c5;

// Para t7, t8, t9:
t7 = atan2(((-a6*s5 + a1)*ax + (-a5*s5 - a2)*ay - a9*az),((-au3 + au4 + au5)*ax+(-au6-au7+au8)*ay+(au9+a10)*az));
vx = pow((-au3 + au4 + au5)*ax + (-au6-au7+au8)*ay+(au9+a10)*az,2) + pow(((-a6*s5+a1)*ax+(-a5*s5-a2)*ay-a9*az),2);
s8 = sqrt(vx);
c8 = (-(-au10-s6*a3-c23*auxa)*ax-(-au11+s6*a4-s23*auxa)*ay-(-aur1+aur2)*az);
t8 = atan2(s8,c8);

t9=atan2((-(-au10-s6*a3-c23*auxa)*nx-(-au11+s6*a4-s23*auxa)*ny-(-aur1+aur2)*nz),((-au10-s6*a3-c23*auxa)*sx+(-
au11+s6*a4-s23*auxa)*sy+(-aur1+aur2)*sz));

// Función que genera el movimiento de LapBot1.

        // para r1
        nodor1->setPosition(0,0,r1*40);

        // para t2
        nodoBrazo->setOrientation(Quaternion(Radian(t2),Vector3( 0,0,1)));//-

        // para t3
        nodoAntebrazo->setOrientation(Quaternion(Radian(t3),Vector3(0,0,1)));//-

        // para t5
        nodot5->setOrientation(Quaternion(Radian(t5),Vector3(0,0,1)));

        // para t6
        nodot6->setOrientation(Quaternion(Radian(t6),Vector3(0,0,1)));

        // para t7
        nodot7->setOrientation(Quaternion(Radian(t7),Vector3(0,0,1)));

        // para t8
        nodot8->setOrientation(Quaternion(Radian(t8),Vector3(0,0,1)));

        // para t9
        nodot9->setOrientation(Quaternion(Radian(t9),Vector3(0,0,1)));

        t2aux=t2;
        t3aux=t3;

// Actualizacion de posición de cuerpo envolvente
pos= nodopinza->_getDerivedPosition();
Body->setGlobalPosition(pos);

//ALGORITMO DE FUERZA F=-KX
if(collision==true)
{
    mDebugText = "--- COLISION DETECTADA ---";
    pactual=mDevice->getPosition();
    Vector3 X=pactual-pante;
    Vector3 F=100*X;
            force[0]=F.x;
```

```
                force[1]=F.y;
                force[2]=F.z;

                hdlSetToolForce(force);
                //mDevice->_applyForces(F, Vector3::ZERO);
                i=i+1;

                }
                else{
                if(j<10)j=j+1;

                if(j>=9){ mDebugText = "";
                j=0;}

                pante=mDevice->getPosition();
                }

                if(i>=7) {
                collision=false;
                i=0;}

                mDebugRenderer->clear();
                mDebugRenderer->draw();

        }

//Para el movimiento de la camara
Ogre::Vector3 lastMotion = mTranslateVector;
            if (mTimeUntilNextToggle >= 0)
            {
                        mTimeUntilNextToggle -= evt.timeSinceLastFrame;
            }
// Mover cerca de 100 unidades por segundo
            mMoveScale = mMoveSpeed * evt.timeSinceLastFrame;
// Toma diez segundos la rotación completa
            mRotScale = mRotateSpeed * evt.timeSinceLastFrame;

        mRotX = 0;
        mRotY = 0;
        mTranslateVector = Ogre::Vector3::ZERO;
        if (!processUnbufferedKeyInput(evt))
        {
            return false;
        }
        if (!processUnbufferedMouseInput(evt))
        {
            return false;
        }
        if (mTranslateVector == Ogre::Vector3::ZERO)
        {
            mCurrentSpeed -= evt.timeSinceLastFrame * 0.3;
            mTranslateVector = lastMotion;
        }
        else
        {
            mCurrentSpeed += evt.timeSinceLastFrame;
        }
        // Limitar el movimiento de la camara
        if (mCurrentSpeed > 1.0)
        {
            mCurrentSpeed = 1.0;
        }
        if (mCurrentSpeed < 0.0)
        {
            mCurrentSpeed = 0.0;
        }
    mTranslateVector *= mCurrentSpeed;
    moveCamera();
    return true;
  }
```

```
    bool frameEnded(const FrameEvent& evt)
  {
    if (mShutdownRequested)
    {
      return false;
    }
    else
    {
      return OgreHapticsDemoFrameListener::frameEnded(evt);
    }
  }
protected:
        bool mShutdownRequested;
        NxOgre::TimeController* mTimeController;
};
#endif
```

Anexo C. Manual de Usuario de la Aplicación 3D para LapBot

La aplicación que se presenta a continuación fue desarrollada para la visualización tridimensional del robot para laparoscopia llamado LapBot, que sigue las trayectorias realizadas por un usuario a través de la interfaz háptica Novint Falcon pasando por un punto de incisión. Además esta aplicación muestra la detección de colisiones, con realimentación de fuerza, entre el efector final del robot y el abdomen simulado del paciente.

C.1. Requerimientos

Para que la aplicación funcione correctamente, el computador donde se va a instalar debe tener:

1. Windows XP, Windows Vista. (No se ha probado con Linux).
2. OpenGL o Direct3D (Normalmente OpenGL siempre está instalado).
3. 150MB o más de espacio en el disco duro.
4. 1GB o más de memoria RAM.
5. Procesador de 1.8GHz o mejor.
6. Preferiblemente una tarjeta aceleradora de video (funcionan mejor con Direct3D).

Los requerimientos se han determinado de acuerdo a las pruebas que se han realizado en algunos computadores, sin embargo es posible que funcione en un computador con características similares.

C.2. Instalación

Después de verificar los requerimientos del sistema, proceda a:

1. Instalar los drivers del dispositivo háptico que vienen en el CD o desde la pagina principal de Novint Technologies http://home.novint.com/support/download.php

2. Instalar el Software del sistema de PhysX para el correcto funcionamiento del motor físico.
3. Ejecutar el archivo con el nombre LapBotSimulator.exe y empezar a probar la aplicación. Si tiene alguna dificultad comuníquese con las autoras sandra.pusil@gmail.com o kvctoria_151@hotmail.com.

C.3. Manejo

La primera ventana que se abre al correr el ejecutable del programa (Figura C.1), es automáticamente generada por Ogre para que el usuario tenga la posibilidad de escoger, entre otras cosas, la API gráfica o subsistema de generación gráfica (OpneGL o Direct3D[9]), la resolución de la ventana, el dispositivo físico para aceleración de video y si desea que el simulador se muestre en pantalla completa.

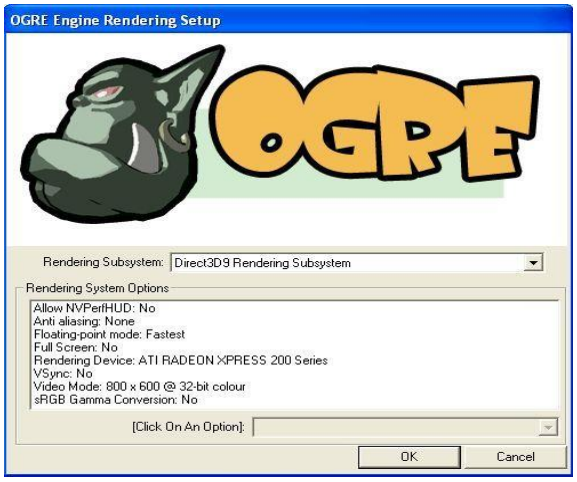

Figura C.1 Interfaz Grafica

Después se abre otra ventana donde se puede seleccionar el dispositivo háptico con el cual se va a trabajar en este caso la interfaz háptica Novint Falcon como se muestra en la Figura C.2.

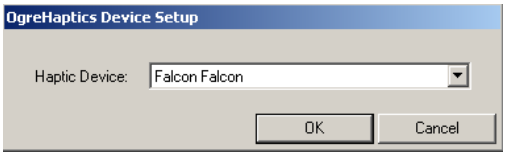

Figura C.2 Selección de dispositivo háptico

[9] Direct3D: utilizado para el procesamiento y la programación de gráficos en tres dimensiones (una de las características más usadas de DirectX).

Una vez escogido el dispositivo háptico se muestra en pantalla la escena quirúrgica donde se encuentran el robot, el abdomen y la sala de operaciones, como se puede observar en la Figura C.3:

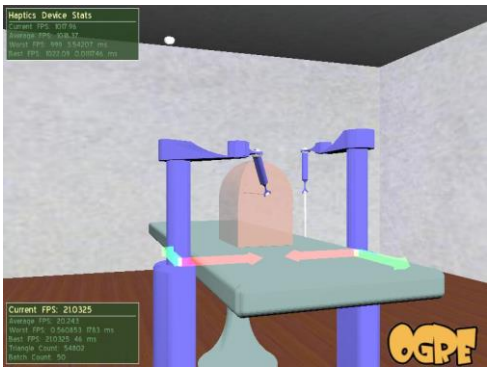

Figura C.3 Escena Quirúrgica

El sistema le permite interactuar con el ambiente 3D por medio del teclado y ratón del computador, un resumen de funciones de teclas y demás se presenta en la Tabla C.5.1.

Tabla C.5.1 Funciones del teclado y ratón

Tecla/ Ratón	***Función***
A	Mueve la cámara hacia la izquierda
S	Mueve la cámara hacia atrás
D	Mueve la cámara hacia la derecha
W	Mueve la cámara hacia delante
R	Muestra los planos
Q	Sale de la aplicación
F	Muestra u oculta la velocidad de simulación
Esc	Sale de la aplicación
P	Muestra la posición y orientación de la cámara
PgUp	Mueve la cámara hacia arriba
PgDown	Mueve la cámara hacia abajo
Flecha arriba	Mueve la cámara hacia delante
Flecha abajo	Mueve la cámara hacia atrás
Flecha derecha	Rota la cámara a la derecha
Flecha izquierda	Rota la cámara a la izquierda
Ratón	Permite navegar por toda la escena