

# **Interfaz Háptica Tipo Guante Con Realimentación Vibratoria**



**Mónica Rocío Díaz Tribaldos  
José Manuel Escobar Ocampo**

**Director: PhD Oscar Andrés Vivas Albán**

*Universidad del Cauca*

**Facultad de Ingeniería Electrónica y Telecomunicaciones  
Departamento de Electrónica, Instrumentación y Control  
Ingeniería en Automática Industrial  
Popayán, Junio de 2013**

# **Interfaz Háptica Tipo Guante Con Realimentación Vibratoria**



Documento Final de Trabajo de Grado para optar al título de  
Ingeniero en Automática Industrial

**Mónica Rocío Díaz Tribaldos**  
**José Manuel Escobar Ocampo**

Director: PhD Oscar Andrés Vivas Albán

*Universidad del Cauca*

**Facultad de Ingeniería Electrónica y Telecomunicaciones**  
**Departamento de Electrónica, Instrumentación y Control**  
**Ingeniería en Automática Industrial**  
**Popayán, Junio de 2013**

## TABLA DE CONTENIDO

INTRODUCCION.....	1
1. INTERFACES HAPTICAS.....	3
1.1 INTERFACES HAPTICAS.....	4
1.1.1 Joystick Sandpaper.....	4
1.1.2 Touch Master.....	4
1.1.3 Master Finger 2.....	5
1.1.4 PHANTOM.....	5
1.2 GUANTES HAPTICOS.....	6
1.2.1 Dextrous HandMaster Exoeskeleton.....	6
1.2.2 CyberGlove II.....	7
1.2.3 CyberTouch.....	7
1.2.4 CyberGrasp.....	8
1.2.5 5DT Data Glove 5 Ultra.....	8
2. HARDWARE DEL GUANTE CONSTRUIDO.....	10
2.1 SENSADO.....	10
2.1.1 Sensor de efecto hall.....	10
2.1.2 Sensor de flexión ( <i>flex sensor</i> ).....	11
2.2 REALIMENTACION.....	14
2.2.1 Vibrador tipo moneda.....	15
2.3 COMUNICACIÓN.....	15
2.3.1 Microcontroladores.....	15
2.3.2 Tarjeta de adquisición Arduino.....	16
2.4 HARDWARE ADICIONAL.....	16
2.4.1 Circuito de adecuación.....	16
2.4.2 Piezas en acrílico.....	18

2.4.3 Guante háptico .....	19
3.    DESCRIPCION DE LAS HERRAMIENTAS SOFTWARE UTILIZADAS .....	20
3.1    ROS .....	20
3.1.1 Conceptos básicos de ROS .....	21
3.1.2 Instalación de ROS .....	23
3.2    PAQUETES DE ROS UTILIZADOS .....	26
3.2.1 Gazebo .....	26
3.2.2 Rosserial .....	26
3.2.3 URDF .....	27
3.2.4 Shadow Robot .....	30
3.2.5 Instalación e inicialización de los paquetes de ROS utilizados .....	31
3.3    RECONOCIMIENTO DE SHADOW HAND .....	35
3.4    QT .....	38
3.5    VTK .....	39
3.6    VCollide .....	39
4.    DESARROLLO DEL SOFTWARE PARA VIRTUALTOUCH .....	40
4.1    CARACTERIZACIÓN DE LOS FLEX SENSOR .....	40
4.2    DESARROLLO DEL SOFTWARE EN ROS .....	44
4.2.1 Adquisición de datos .....	45
4.2.2 Publicadores y suscriptores para Shadow Hand .....	49
4.2.3 Sistema de detección de colisiones .....	51
4.2.4 Adecuación del entorno virtual .....	54
4.3    DESARROLLO DEL SOFTWARE EN QT/VTK .....	56

4.3.1	Desarrollo de la interfaz de usuario .....	56
4.3.2	Comunicación de la interfaz con Arduino .....	59
4.3.3	Detección de colisiones.....	60
4.3.4	Deformación de objetos .....	63
5.	CONCLUSIONES Y TRABAJOS FUTUROS .....	66
5.1	CONCLUSIONES .....	66
5.2	TRABAJOS FUTUROS.....	66
6.	REFERENCIAS .....	68

## LISTADO DE TABLAS

Tabla 4.1. Medición de los <i>flex sensor</i> (resistencia – bits).....	41
Tabla 4.2. Toma de datos de <i>flex sensor</i> – variación en rango de 0° a 90°..	42
Tabla 4.3. Medición <i>flex sensor</i> para cada dedo. ....	43

## LISTADO DE FIGURAS

Figura 1.1. Sandpaper en uso.....	4
Figura 1.2. Touch Master de EXOS. ....	4
Figura 1.3. MasterFinger-2.....	5
Figura 1.4. PHANTOM Omni (a), PHANTOM desk (b), PHANTOM Premium 3.0 6DOF (c).....	6
Figura 1.5. Dextrous Hand Master Exoskeleton. ....	6
Figura 1.6. CyberGlove II. ....	7
Figura 1.7. CyberTouch. ....	8
Figura 1.8. CyberGrasp.....	8
Figura 1.9. 5DT Data Glove 5 Ultra. ....	9
Figura 2.1. Sensores de Efecto Hall. ....	11
Figura 2.2. <i>Flex sensor</i> . ....	12
Figura 2.3. Variación de la resistencia en función del ángulo. ....	12
Figura 2.4. Divisor de Voltaje para el <i>flex sensor</i> . ....	13
Figura 2.5. Circuito de accionamiento. ....	13
Figura 2.6. Amplificador inversor.....	14
Figura 2.7. Motor de vibración.....	15
Figura 2.8. Tarjeta Arduino Mega 2560. ....	16
Figura 2.9. Circuito de adecuación.....	17
Figura 2.10. Diagrama de conexión <i>flex sensor</i> a tarjeta Arduino.....	17
Figura 2.11. Piezas en Acrílico. ....	18
Figura 2.12. Impresora 3D <i>systems Projet HD 3000</i> . ....	18
Figura 2.13. Primer prototipo VirtualTouch.....	19
Figura 2.14. Versión final VirtualTouch. ....	19
Figura 3.1. Funcionamiento básico de temas y nodos.....	22
Figura 3.2. Pantalla de inicio de Ubuntu.....	24
Figura 3.3. Pantalla de búsqueda del terminal. ....	24
Figura 3.4. Instalación ROS. ....	25
Figura 3.5. Modelo simple de un robot con URDF.....	30
Figura 3.6. Shadow Hand.....	30
Figura 3.7. Entorno gráfico de Gazebo. ....	31

Figura 3.8. IDE de Arduino. ....	32
Figura 3.9. Shadow Hand en Gazebo. ....	35
Figura 3.10. Shadow Robot en sus dos versiones. ....	35
Figura 3.11. Anatomía de la mano. ....	36
Figura 4.1. Gráfica de datos obtenidos por el <i>flex sensor</i> en la variación de 0° a 90°. ....	43
Figura 4.2. Rostopic list una vez iniciado roserial. ....	48
Figura 4.3. rostopic echo dedo índice. ....	49
Figura 4.4. Gráfica de comunicación entre Shadow Hand y Arduino. ....	51
Figura 4.5. Entorno virtual para VirtualTouch. ....	55
Figura 4.6. Prueba usando VirtualTouch en ROS. ....	56
Figura 4.7. Interfaz creada en QT Creator. ....	57
Figura 4.8. Pipeline de VTK para renderizado. ....	57
Figura 4.9. Interfaz VirtualTouch en VTK/QT. ....	59
Figura 4.10. Eje de coordenadas de la mano 3D en Blender. ....	60
Figura 4.11. Clase DeteccionColisiones. ....	61
Figura 4.12. Diagrama de Flujo de detección de colisiones. ....	61
Figura 4.13. Visualización no contacto. ....	62
Figura 4.14. Visualización de contacto. ....	63
Figura 4.15. Estructura del motor de deformación. ....	63
Figura 4.16. Prueba VirtualTouch con Deformación. ....	65
Figura 4.17. Utilización de VirtualTouch con QT/VTK. ....	65





## INTRODUCCION

Los últimos avances en la robótica han permitido su aplicación en casi todas las ramas de la ciencia, incluyendo la medicina, en donde cada día ocupa un papel más importante. En la actualidad es una excelente herramienta de asistencia al momento de realizar cirugías complejas, o en aquellos casos donde una falla por parte del ser humano es muy probable, como también la sustitución de las extremidades del cuerpo.

Dentro del campo de la robótica médica hay numerosas herramientas disponibles para el uso en medicina. Entre dichas herramientas se encuentran las llamadas interfaces hápticas, que buscan permitir la interacción entre personas y máquinas. La investigación en este campo busca generar estímulos mecánicos que puedan producir una respuesta a la interacción virtual que tiene el usuario en forma de una respuesta física sensorial.

Los guantes hápticos hacen parte del desarrollo y el estudio de los dispositivos de realidad virtual. Básicamente un guante háptico es un mecanismo que se adapta a la forma de la mano humana. Esta clase de interfaz permite el fácil movimiento de los dedos, los cuales deben tener dispositivos electrónicos o mecánicos de tal forma que las sensaciones de tomar un objeto o tocar una superficie en un entorno virtual, sean similares a las sensaciones reales.

De otra parte la tecnología médica ha producido en los últimos años dispositivos para realizar la tomografía axial computada (TAC), también conocida como tomografía computada (TC), la cual permite obtener imágenes y cortes en tres dimensiones de los órganos del cuerpo humano. Esta tecnología se ha convertido en gran ayuda para los médicos en el diagnóstico de enfermedades de los pacientes, ya que se puede observar con gran detalle el interior del cuerpo humano, sin llegar a explorarlo directamente.

De acuerdo a los dos avances tecnológicos anteriormente descritos, y sabiendo que las interfaces hápticas están apenas en fase de desarrollo, pretendiendo que en un futuro los médicos aparte de la tomografía computarizada puedan tener un elemento de juicio adicional en su diagnóstico, es decir que además de “observar” el interior del cuerpo humano, ellos puedan llegar a “tocarlo” de manera virtual. Se necesita aclarar que esto incluiría un gran avance en el procesamiento de imágenes provenientes del TAC con el fin de lograr diferenciar los más pequeños detalles de las características encontradas internamente. De igual manera implicaría disponer

de guantes hápticos de gran sensibilidad que logren proporcionar al médico la sensación de tocar los detalles nombrados anteriormente.

Tratando de vislumbrar las aplicaciones futuras de estas nuevas tecnologías, y con el fin de contribuir a la compleja problemática de lo que es un diagnóstico médico, el presente trabajo se plantea como objetivo diseñar un guante háptico con realimentación vibratoria, y una interfaz para el uso de la misma, donde se pueda percibir el contacto con algún objeto o superficie, utilizando herramientas de software libre como ROS, QT, VTK y VCollide. Las herramientas QT, VTK y VCollide se usaron debido a que durante el desarrollo del trabajo ROS no tenía implementado la deformación de objetos por lo cual con el fin de completar y cumplir con los objetivos del proyecto se tomó como herramienta alterna el desarrollo de una aplicación distinta con QT, VTK y VCollide que permitiera realizar deformación. De igual manera cabe mencionar que este será el primer guante desarrollado en la Universidad del Cauca y el primer desarrollo con ROS en pregrado de Ingeniería en Automática Industrial.

El desarrollo del documento va de la siguiente manera. En el capítulo 1, se presentan las interfaces hápticas encontradas en el mercado, incluyendo las tipo guante que existen en la actualidad. En el capítulo 2, se explicara el hardware utilizado en la construcción del guante VirtualTouch. En el capítulo 3, se describirán las herramientas software utilizadas. El capítulo 4 contendrá el desarrollo software/hardware del proyecto. El capítulo 5 tendrá conclusiones referentes al trabajo realizado y planteamiento de trabajos futuros, y finalmente en el capítulo 6 las referencias bibliográficas empleadas.

Cabe mencionar que para la finalización exitosa del proyecto se contó con el apoyo académico y material del PhD. José María Sabater Navarro, del Departamento de Ingeniería de Sistemas y Automática de la Universidad Miguel Hernández de Elche, España, en donde se realizó una pasantía de 3 meses en la cual se construyó el guante, soportado económicamente por el proyecto OpenSurg.

## 1. INTERFACES HAPTICAS

El hombre diariamente está expuesto a las sensaciones hápticas, sensaciones referidas al sentir, tocar o palmar objetos, en este también se encuentran inmersos sentidos de vista y audio. En la interacción hombre-computador requerida en un entorno virtual, generalmente se hace uso de las facultades visuales (a través de una pantalla) y auditivas (parlantes) para la realimentación de información de una escena o entorno, sin embargo los niveles de inmersión del usuario pueden mejorarse proporcionando más realismo en la interacción y es de ahí donde reside la necesidad de las interfaces hápticas o kinestésicas, ya que son dispositivos bidireccionales, en donde sus principales entradas/salidas son el desplazamiento y la fuerza.

La información sensorial se puede distinguir de dos maneras la táctil y la kinestésica. La primera hace referencia a los receptores de tacto en la piel que dan la información de la geometría, textura y viscosidad de los objetos o elementos que se manipulan, y la kinestésica que es la que nos proporciona la información de posición, y peso, completando así la sensación que produce, tocar o manipular un objeto [1].

Con “interfaz háptica” aludimos a aquellos dispositivos que permiten al usuario “tocar”, sentir o manipular objetos simulados en entornos virtuales e interactuar con sistemas teleoperados. En algunos casos de simulaciones realizadas en entornos virtuales ha sido suficiente emplear *displays* y dispositivos de sonido para provocar en el usuario inmersión en el entorno mediante imágenes o sonidos. Además de provocar en el usuario inmersión se hace necesario proporcionar también la posibilidad de interactuar un poco más con el entorno en donde haya más interacción bidireccional mediante las interfaces hápticas.

Algunos de los principales medios de desempeño y aplicación de las interfaces hápticas se encuentra en:

- *Medicina*: Los simuladores quirúrgicos de entrenamiento, micro robots para cirugías mínimamente invasivas, entre otras.
- *Educacional*: Experimentación de fenómenos a escalas nano y macro.
- *Entretenimiento*: Videojuegos y simuladores que permiten al usuario sentir y manipular objetos virtuales.
- *Industria*: Integración de las interfaces hápticas en los sistemas CAD de tal forma que el usuario pueda manipular libremente los componentes de un conjunto en un entorno inmersivo.

- *Artes Gráficas*: Exhibiciones virtuales de arte, museos y otras.

## 1.1 INTERFACES HAPTICAS

### 1.1.1 Joystick Sandpaper

El sistema Sandpaper desarrollado en 1990 cuenta con dos grados de libertad. La posición del joystick es reportada al software que computa las fuerzas apropiadas a los motores del joystick, permitiendo así la exploración de superficies rugosas mediante una simple regla de fuerzas de contacto que son proporcionales al gradiente local de la superficie con textura (Figura 1.1) [2].



Figura 1.1. Sandpaper en uso.  
Fuente: [2].

### 1.1.2 Touch Master

Esta interfaz fue desarrollada por EXOS en 1993, (Figura 1.2), Es una interfaz táctil que nos permite la simulación de cada uno de los cinco dedos, los actuadores son de tipo electromagnético, estos dan una realimentación vibratoria, que suministran una frecuencia entre 210-240 Hz en amplitud constante [1].



Figura 1.2. Touch Master de EXOS.  
Fuente: [1].

### 1.1.3 Master Finger 2

Es una interfaz háptica modular en donde cada dedo se administra de una forma independiente. La estructura mecánica y el controlador son compartidos por todos los módulos. El diseño del módulo mecánico se basa en una estructura serie-paralelo, que otorga un área de trabajo amplia y con una inercia pequeña. La manipulación es cómoda gracias a la base que esta tiene. El MasterFinger-2 (Figura 1.3) se compone de dos módulos, colocados de tal manera que el índice y el pulgar sean los dedos a utilizar, permitiendo así la interacción del usuario con los entornos virtuales de una manera fácil y cómoda [3].

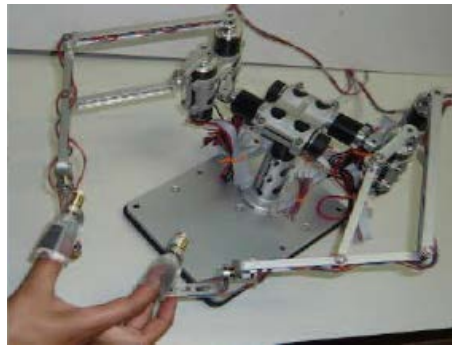


Figura 1.3. MasterFinger-2.  
Fuente: [3].

### 1.1.4 PHANTOM

La línea Sensable, ha creado el dispositivo háptico PHANTOM, que permite a los usuarios tocar y manipular objetos virtuales, basándose en las fuerzas. Existen diferentes modelos de PHANTOM mostrados a continuación [4]:

*PHANTOM Omni*: Es el más económico de la línea, presenta un diseño más portable, como también una instalación y uso fácil, el movimiento giratorio de la muñeca es posible, cuenta con seis grados de libertad (Figura 1.5 (a)) [5].

*PHANTOM Desktop*: Esta versión (Figura 1.5 (b)) de PHANTOM es una versión de escritorio más asequible e ideal para la investigación háptica, cuenta con precisa información de posicionamiento y una buena realimentación de fuerza. Posee seis grados de libertad [5].

*PHANTOM Premium 3.0/6DOF*: Este dispositivo (Figura 1.5 (c)) permite a los usuarios explorar áreas de aplicación que requieran fuerza de respuesta

sobre todos los grados de libertad, presenta una amplia área de trabajo, como rotacional [6].

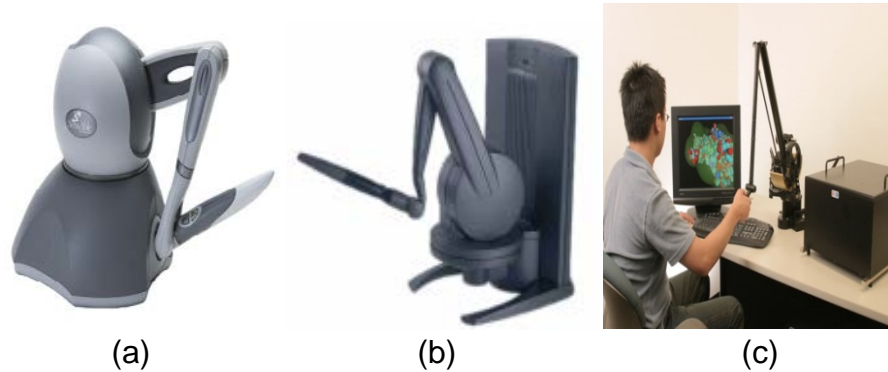


Figura 1.4. PHANTOM Omni (a), PHANTOM desk (b), PHANTOM Premium 3.0 6DOF (c).  
Fuentes: [5], [6].

## 1.2 GUANTES HAPTICOS

### 1.2.1 Dextrous HandMaster Exoskeleton

El pionero en la creación de interfaces hápticas es el Instituto Tecnológico de Massachusetts (MIT), con el dispositivo háptico de configuración exoesquelética Dextrous hand master exoeskeleton (Figura 1.5). El 'dextrous handMaster' (DHM) es un exoesqueleto que se ajusta a los dedos mediante unas cintas de velcro, en cada falange o junta cuenta con un sensor de efecto Hall que proporciona la información del ángulo formado entre las juntas. No usa señales ópticas o eléctricas, el DHM usa articulaciones mecánicas para seguir el movimiento de la mano, y la desviación radial-cubital (movimiento de lado a lado) de cada dedo [7].

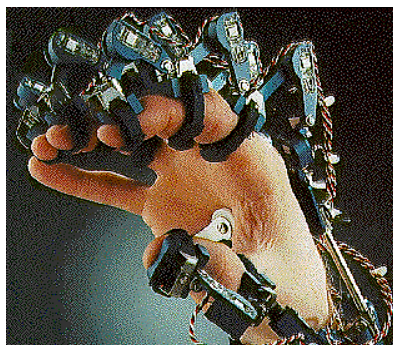


Figura 1.5. Dextrous Hand Master Exoskeleton.  
Fuente: [8].

### 1.2.2 CyberGlove II

El CyberGlove (Figura 1.6) es un guante electrónico inalámbrico, que transforma los movimientos de los dedos y la mano en tiempo real para ser utilizado en aplicaciones de animación, realidad virtual, prototipado digital, estudios biomecánicos, entrenamientos médicos y militares entre otros. El guante cuenta con 18 sensores, dos '*bend sensors*' (sensores de curvatura), dos en cada dedo, cuatro sensores de abducción (*abduction sensors*), además de sensores que miden el movimiento del pulgar, el arco de la palma y la flexión de la muñeca. La resolución de los sensores es menor de un grado, su repetitividad de tres grados, y su linealidad tiene un máximo del 0.6% del rango de la articulación. La tecnología inalámbrica con la que cuenta es de 2.4 GHz, su batería dura 3 horas, el rango de funcionamiento son 9.1400 metros del puerto USB donde se da la conexión inalámbrica [9].



Figura 1.6. CyberGlove II.  
Fuente: [9].

### 1.2.3 CyberTouch

El CyberTouch (Figura 1.7) proporciona una respuesta táctil para el CyberGlove, descrito anteriormente (Sección 1.2.1). Los estimuladores táctiles se unen a cada una de las puntas de los dedos y a la palma del usuario para proporcionar una vibración mediante impulsos o una vibración continua, su frecuencia de vibración oscila entre 0 y 125 Hz, los actuadores se pueden programar individualmente y de esta forma brindar un mayor nivel de realimentación [9].



Figura 1.7. CyberTouch.  
Fuente: [9].

#### 1.2.4 CyberGrasp

El sistema CyberGrasp (Figura 1.8) es un sistema de realimentación de fuerzas resistivas en dedos y mano, permitiendo así la tele-manipulación de objetos, logrando que los usuarios sean capaces de sentir el tamaño y la forma de los objetos generados por ordenador en un entorno 3D. Su diseño presenta una estructura liviana, el exoesqueleto se ajusta perfectamente al CyberGlove (Sección 1.2.1) [9].



Figura 1.8. CyberGrasp.  
Fuente: [9].

#### 1.2.5 5DT Data Glove 5 Ultra

El 5DT Data Glove 5 Ultra (5DT 5) (Figura 1.9) está diseñado para satisfacer las exigencias de la animación profesional, es cómodo, de fácil uso y pequeño. Los controladores pueden ser utilizados en diversas aplicaciones, presenta una alta calidad en los datos por lo que es ideal para la animación en tiempo real, se compone por un *flex sensor* en cada dedo, y su comunicación es vía USB. [10]





Figura 1.9. 5DT Data Glove 5 Ultra.  
Fuente: [10].

## 2. HARDWARE DEL GUANTE CONSTRUIDO

El desarrollo del guante háptico VirtualTouch, conlleva una estructura física que se encuentra dividida en cuatro partes.

- Sensado.
- Realimentación.
- Comunicación.
- Hardware adicional.

Cada uno de estos subsistemas contiene una parte fundamental, la cual fue seleccionada gracias a los requerimientos de presupuesto y simplicidad al momento del desarrollo del guante háptico. Siendo este el primer prototipo a construir, se tuvieron en cuenta dispositivos como sensores de efecto Hall, *flex sensors*, vibradores tipo moneda, microcontroladores y la tarjeta de adquisición Arduino.

### 2.1 SENSADO

Para el sensado de los movimientos de los dedos de la mano es necesario obtener valores confiables y bastante acertados, ya que de esto depende que el movimiento simulado dentro del entorno gráfico de VirtualTouch sea lo más cercano a la realidad. Para esto se estudió varios tipos de sensores que podían ser útiles para la construcción del guante háptico. Dentro de estas posibilidades y rigiéndose por los requerimientos y características del guante, se pensó en varias alternativas nombradas a continuación:

#### 2.1.1 Sensor de efecto hall

El sensor de efecto Hall (Figura 2.1) es un tipo de sensor que se sirve del efecto Hall para la medición de campos magnéticos o para determinar cierta posición. El efecto Hall relaciona la tensión entre dos puntos de un material conductor o semiconductor con un campo magnético a través de un material. Si fluye corriente por un sensor Hall y se aproxima a un campo magnético que fluye en dirección vertical al sensor, entonces el sensor crea un voltaje saliente proporcional al producto de la fuerza del campo magnético y de la corriente. Si se conoce el valor de la corriente, entonces se puede calcular la fuerza del campo magnético; si se crea el campo magnético por medio de corriente que circula por una bobina o un conductor, entonces se puede medir el valor de la corriente en el conductor o bobina.

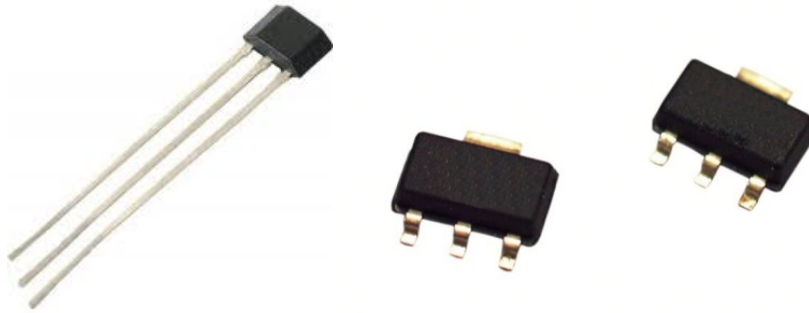


Figura 2.1. Sensores de Efecto Hall.  
Fuente: [11].

Este tipo de sensores son útiles para la medición de la posición de las falanges de cada dedo, para lo cual sería necesario un sensor por cada falange, es decir 3 sensores por dedo y 15 por mano. Aunque es un dispositivo económico cumpliendo con los requerimientos de presupuesto, la utilización de estos sensores no se hace fácil al tener que fijar un campo electromagnético fijo para la medición de los movimientos, lo que no es una tarea sencilla, además son dispositivos muy vulnerables ante el ruido eléctrico y electromagnético, lo cual podría llegar a significar un contratiempo al momento de sensar los movimientos de la mano. Por estos motivos este tipo de sensor fue descartado para el desarrollo del dispositivo.

### 2.1.2 Sensor de flexión (*flex sensor*)

El sensor de flexión o *flex sensor* (Figura 2.2), es un dispositivo pasivo resistivo, que varía el valor de su resistencia según el ángulo de flexión que se le genere, es un dispositivo bidireccional, es decir que varía su valor de resistencia en ángulos comprendidos entre  $0^\circ$  y  $180^\circ$ . Este tipo de sensores son fáciles de utilizar y bastante resistentes ante la manipulación.

Los *flex sensors* consisten en sí de varios sensores que cambian su resistencia en función de la cantidad de curvatura aplicada. La variación de la resistencia del *flex sensor* cambia dependiendo si las pastillas de metal están en el exterior o en el interior de la curva. A mayor sea la curva, mayor será el valor de la resistencia. El *flex sensor* tiene una resistencia nominal de  $10\text{ K}\Omega$ , cuando está totalmente recto, el rango de variación está entre los  $60\text{ K}\Omega$  y  $110\text{ K}\Omega$  como se muestra en la Figura 2.3. El rango de temperatura de trabajo está entre  $-35^\circ\text{C}$  y  $80^\circ\text{C}$ .



Figura 2.2. *Flex sensor*.

Fuente: [12].

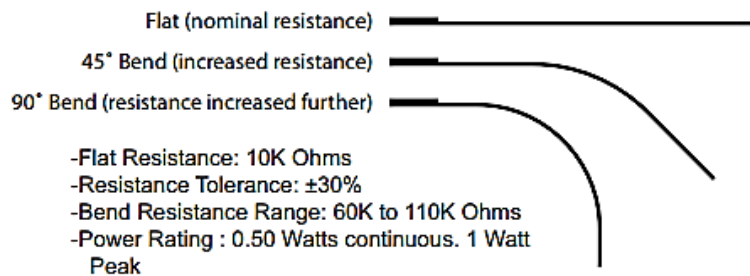


Figura 2.3. Variación de la resistencia en función del ángulo.

Fuente: [12].

El *flex sensor* se utiliza en guantes de juego, guantes como el CyberGlove, CyberGrasp, controles de autos, aparatos de medición, instrumentos musicales, palancas de mando y más.

Después de analizar los tipos de sensores se opta por utilizar el *flex sensor* ya que provee un buen rango dinámico de variación. La mayor ventaja que presenta este sensor al momento de la implementación del modelo desarrollado es que es posible el uso de un solo sensor por dedo, ya que por medio de cálculos matemáticos se puede simular el ángulo del movimiento de cada una de las falanges, en nuestro caso la falange proximal de cada dedo permite el movimiento de las otras dos falanges (media y distal).

El fabricante recomienda trabajar tres configuraciones de circuitos que se describen a continuación:

- **Divisor de Voltaje:**

El *flex sensor* como divisor de voltaje aumenta el voltaje de salida con la deflexión del sensor. Es el circuito básico (Figura 2.4) más usado para el uso de estos, el *flex sensor* es conectado en serie con una resistencia  $R_2$ , el voltaje de entrada  $V_{in}$  que puede ser, o no, la tensión de la fuente de alimentación, conectada a  $R_1$  (*flex sensor*). La resistencia  $R_2$ , deberá ir a tierra. El voltaje de salida es el voltaje  $V_{in}$  multiplicado por la división de  $R_1$  sobre la suma de las resistencias, como se ve en la ecuación de

la Figura 2.4.

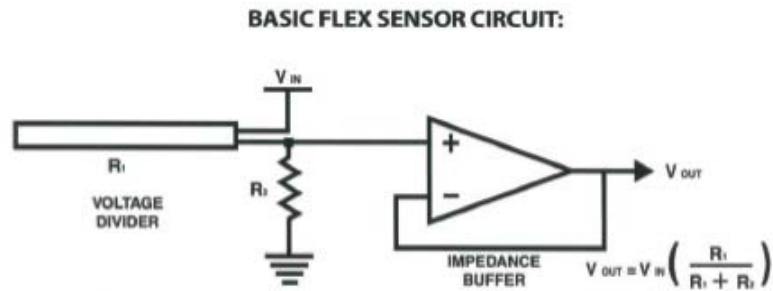


Figura 2.4. Divisor de Voltaje para el *flex sensor*.  
Fuente: [12].

- **Circuito de accionamiento:**

Este circuito de accionamiento funciona como un *switch*, si el voltaje de entrada en el borne positivo es mayor que la tensión conectada al borne negativo, la salida  $V_{out}$  será positiva. En caso contrario, la salida tendrá un voltaje negativo. De esta manera se puede utilizar el *flex Sensor* como un interruptor, sin pasar a través de un microcontrolador, el circuito propuesto está en la Figura 2.5.

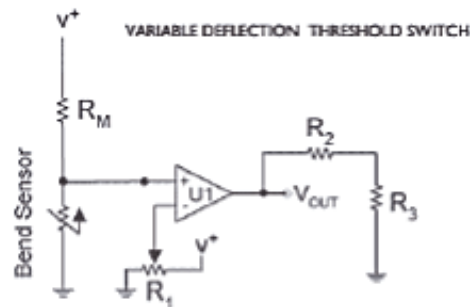


Figura 2.5. Circuito de accionamiento.  
Fuente: [12].

- **Amplificador inversor:**

En este circuito la entrada positiva está conectada a tierra, y la señal se aplica a la entrada negativa a través del *flex sensor*, con realimentación desde la salida a través de  $R_G$ . Se usa en situaciones donde el rango de trabajo en grados de deflexión es pequeño. En la Figura 2.6 se observa el circuito.

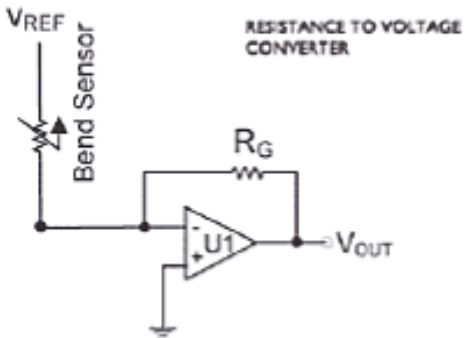


Figura 2.6. Amplificador inversor.  
Fuente: [12].

Una vez revisado el funcionamiento de cada uno de los circuitos se escoge la configuración divisor de voltaje debido al propio funcionamiento del *flex sensor*, sin implementar circuitos complejos que aumenten los costos. Los *flex sensors* tienen un valor aproximado de 44.000 pesos cada uno.

## 2.2 REALIMENTACION

El guante deberá contar con un medio de realimentación por lo que se hace necesario utilizar un actuador capaz de brindar la sensación táctil, sensación bastante compleja por todas las terminaciones nerviosas presentes en la piel y explícitamente en las yemas de los dedos. Por ello se implementa un dispositivo de bajo costo que más que simular un contacto con algún objeto y/o superficie fuera capaz de brindar una advertencia de colisión y una de deformación. Para este objetivo se tuvieron en cuenta choques eléctricos pequeños, motores para simular las fuerzas e impedir el movimiento de los dedos o, señales vibratorias.

Para la primera opción referente a los choques eléctricos se apreció que la mayoría de las personas presentan miedo y rechazo a este tipo de señales, por lo que no sería cómodo para el usuario este tipo de realimentación. Además que un mal manejo de estas puede llevar a consecuencias negativas para el usuario. La segunda referente a motores para la simulación de fuerzas se descartó por la complejidad en cuanto al diseño de su implementación ya que sería construir un exoesqueleto como el *CyberGrasp*. Por lo tanto teniendo en cuenta dispositivos existentes como los videojuegos (Play Station, Xbox, Nintendo, y demás) e incluso los celulares, se optó por un sistema de advertencia vibratoria. Finalmente se buscó dentro de la gran variedad de vibradores el que cumpliera con especificaciones de tamaño, ya que estos deberían ser puestos en las yemas de los dedos.

### 2.2.1 Vibrador tipo moneda

Este tipo de micro motores de vibración son motores de imán permanente, generalmente se encuentran con dimensiones de 10mm aproximadamente, operan con una frecuencia de 225 Hz, a un voltaje de 3v y una corriente de 75mA. El motor adquirido es fabricado por *Precision Microdrivers* en Inglaterra, y comercializados por Sparkfun Electronics, en un precio aproximado de 10.000 pesos cada uno (Figura 2.7).



Figura 2.7. Motor de vibración.  
Fuente: [13].

## 2.3 COMUNICACIÓN

Una parte importante en la construcción de este prototipo es la comunicación, en cuanto a la transmisión y recepción de información desde y hacia el guante. Para esto era necesaria una interfaz de adquisición de datos con el fin de recibir la información de los sensores y a su vez poder simular los movimientos de la mano mediante el software de VirtualTouch, así como transmitir los valores de colisión o deformación hacia los actuadores. Para ello se estudiaron varios dispositivos de adquisición de datos como los siguientes:

### 2.3.1 Microcontroladores

El micro controlador es un circuito integrado programable. Es capaz de ejecutar las órdenes grabadas en su memoria. En su interior cuenta con una unidad de procesamiento, una memoria y periféricos entrada/salida. Su programación puede ser escrita en lenguaje ensamblador o en otro lenguaje, que finalmente se traducirá al sistema numérico hexadecimal, gracias a los compiladores y demás herramientas existentes para la programación de microcontroladores. A pesar de ser dispositivos versátiles y de bajo costo, no se tomaron en cuenta, debido al tiempo que conllevaría crear el circuito de acondicionamiento, y las limitaciones de salidas y entradas analógicas que estos poseen.

### 2.3.2 Tarjeta de adquisición Arduino

La tarjeta de adquisición Arduino (Figura 2.8) es una plataforma de hardware libre, basada en una placa con un microcontrolador y un entorno de desarrollo, diseñada para facilitar el uso de la electrónica en proyectos multidisciplinarios.



Figura 2.8. Tarjeta Arduino Mega 2560.  
Fuente: [14].

El hardware consiste en una placa con un microcontrolador Atmel AVR y puertos de entrada/salida. Los microcontroladores más usados son el ATmega168, ATmega328, ATmega1280, ATmega8 y ATmega2560 por su sencillez y bajo costo, que permiten el desarrollo de múltiples diseños. Por otro lado el software consiste en un entorno de desarrollo que implementa el lenguaje de programación Processing/Wirting y el cargador de arranque que corre en la placa.

Gracias a la simplicidad, la robustez, el bajo costo y la versatilidad en su funcionamiento, fue la tarjeta Arduino Mega 2560 la mejor opción para el desarrollo de VirtualTouch.

## 2.4 HARDWARE ADICIONAL

### 2.4.1 Circuito de adecuación

Para la conexión de los dispositivos de sensado, se diseñó una placa PCB para la interconexión de los *flex sensors*, de este modo y con la ayuda del software de *CadSoft Eagle* [15] se diseñó la placa (Figura 2.9).



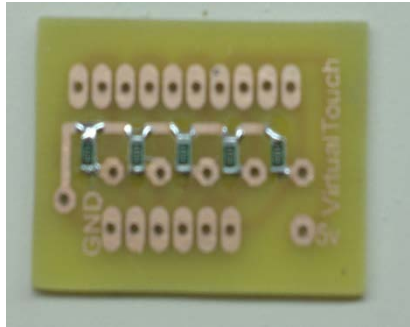


Figura 2.9. Circuito de adecuación.  
Fuente: Propia.

El circuito está compuesto por una sencilla resistencia smd de 10kΩ en serie con el pin GND de los *flex sensor* y la entrada de lectura analógica de la tarjeta Arduino Mega 2560, teniendo aquí implementando un divisor de voltaje. Esto con el fin de acondicionar la señal de entrada analógica de la tarjeta Arduino, las señales de entrada varían entre los 0 y 1023 bits donde 0 bits es 0v y 1023 son 5v. La implementación de este circuito de adecuación fue pensado también para evitar daños de la tarjeta al poderse ver afectada por picos de voltaje superiores e inesperados, se estableció una resistencia de 10kΩ para reducir el valor del voltaje que se entrega a la tarjeta, de este modo la tarjeta Arduino en el caso de tener el sensor en 0° (resistencia de 11.04 KΩ), el voltaje de salida es aproximadamente 2,6v, traducidos a bits son 531, de la misma manera al flexionar el sensor en un ángulo de 90° este varía su resistencia a 14,65 KΩ, lo que da 2,02 v de salida y 413 bits. Una vez obtenidos los valores de operación de los sensores en la tarjeta Arduino se debe proceder a modelar dichos valores para así conseguir las curvas de operación de cada sensor como se mostrará más adelante. El diagrama de conexión de los sensores se puede apreciar de mejor modo en la Figura. 2.10.

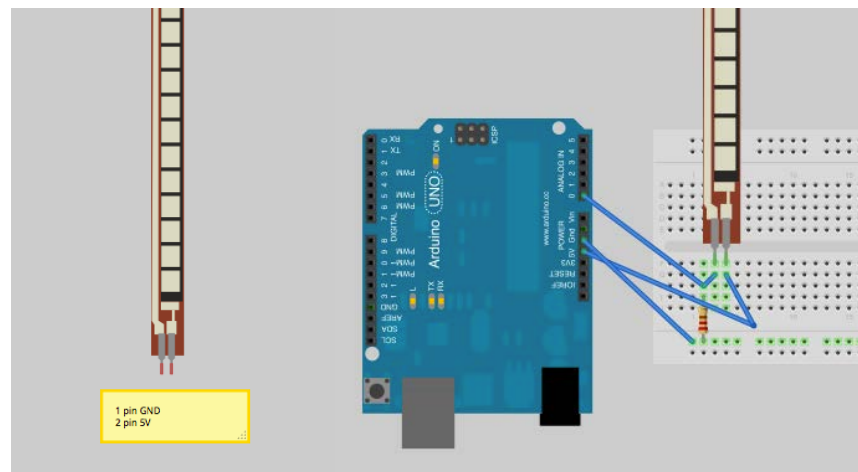


Figura 2.10. Diagrama de conexión *flex sensor* a tarjeta Arduino.  
Fuente: [16].

## 2.4.2 Piezas en acrílico

Se diseñaron unas piezas en el software CAD *Inventor* [17], se diseñó una pieza por cada dedo donde se pudiera ubicar el vibrador. En el caso del dedo pulgar e índice la pieza ha sido diseñada pensando en la deformación y una futura sensación de deslizamiento, y es por ello que esta pieza se diseñó para la ubicación de dos vibradores dentro de ella. En la Figura 2.11 se pueden ver las piezas diseñadas, la primera de izquierda a derecha es la pieza para el dedo pulgar, y junto a este se aprecian las demás piezas para cada dedo. Para los *flex sensors* también se diseñaron unas piezas que permitieran el ajuste en el guante, las cuales se pueden observar en la parte inferior de la Figura 2.11. Estas piezas fueron impresas en la impresora 3D *Systems Projet HD 3000* de la Universidad Miguel Hernández de Elche, España (Figura 2.12).



Figura 2.11. Piezas en Acrílico.  
Fuente: Propia.



Figura 2.12. Impresora 3D *systems Projet HD 3000*.  
Fuente: [18].

### 2.4.3 Guante háptico

Una vez adquiridos todos los componentes se realizó el montaje de las piezas, obteniéndose finalmente lo que se observa en la Figura 2.13. Con este primer prototipo en sus iniciales pruebas de movilidad se notó que había movimientos que no dejaba hacer con mayor naturalidad, presentando poca estabilidad y poco seguimiento a los movimientos reales. Por lo que rápidamente se decidió buscar otro tipo de guante, uno más flexible y más cómodo, que permitiera una buena movilidad de la mano, y finalmente se diseñó una caja en acrílico para guardar los circuitos adicionales y la tarjeta Arduino obteniéndose finalmente el prototipo final de esta primera versión de VirtualTouch (Figura 2.14).



Figura 2.13. Primer prototipo VirtualTouch.  
Fuente: Propia.



Figura 2.14. Versión final VirtualTouch.  
Fuente: Propia.

### 3. DESCRIPCION DE LAS HERRAMIENTAS SOFTWARE UTILIZADAS

Para el desarrollo del proyecto también es indispensable el desarrollo de una interfaz que permita la utilización de VirtualTouch, en donde se puedan sentir las colisiones y lograr la deformación de objetos. Para ello se decidió hacer uso de software libre para así evitar costos en licencias de funcionamiento y cumplir con la filosofía del proyecto OpenSurg al cual está adscrito VirtualTouch. De este modo se desarrolló parte del proyecto sobre el meta sistema operativo ROS, el cual es un sistema operativo para robots que trabaja sobre el sistema operativo Ubuntu. Otra parte del proyecto se realizó con Microsoft Visual Studio 2008 el cual trabaja sobre el sistema operativo Windows, con QT, VTK y Vcollide como librerías de desarrollo.

#### 3.1 ROS

ROS es una infraestructura digital para el desarrollo de software para robots que provee la funcionalidad de un sistema operativo. ROS se desarrolló originalmente en 2007 y fue llamado *switchyard* por el Laboratorio de Inteligencia Artificial de Stanford. Desde 2008, el desarrollo continúa primordialmente en Willow Garage, un instituto de investigación robótico con la ayuda de más de 20 instituciones colaboradoras a nivel mundial.

ROS provee los servicios estándar de un sistema operativo, de este modo permite el control de dispositivos de bajo nivel, la abstracción de hardware, paso de mensajes entre procesos, implementación de funcionalidad común y el mantenimiento de paquetes. Está basado en una arquitectura donde el procesamiento toma lugar en los nodos, los cuales pueden recibir, enviar y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores, entre otros. ROS está orientado para un sistema operativo UNIX, donde Ubuntu (Linux) es el sistema soportado por Willow Garage, aunque se está adaptando a otros sistemas operativos como Fedora, Mac OS X, Debian, Microsoft Windows y otros menos conocidos como Arch, Gentoo, OpenSUSE, Slackware, los cuales en estos momentos son considerados como experimentales.

ROS es software libre con términos de licencia que permite libertad para el uso comercial o investigativo, este tipo de licencia es llamada licencia BSD. Las contribuciones de los paquetes en ROS están bajo una gran variedad de licencias diferentes; los lenguajes de programación implementados en ROS son Python, C++ y Lisp [19].

### 3.1.1 Conceptos básicos de ROS

ROS tiene tres niveles básicos, el nivel de sistemas de archivos, el nivel de computación gráfica, y el nivel comunitario [19].

En el nivel de sistemas de archivos encontramos los recursos del sistema en donde existen *paquetes*, *manifest*, *stacks* (pilas), *mensajes*, y *servicios*.

*Paquetes*: Los paquetes son la unidad principal para organizar el software de ROS, los paquetes pueden contener, nodos que son los procesos de ejecución de ROS, las librerías de ROS, datos, archivos de configuración.

*Manifest (manifiestos)*: los *manifests (manifest.xml)* proporcionan los datos de un paquete, es decir las dependencias y licencias que este maneje.

*Stacks (pilas)*: Los stacks son conjuntos de paquetes que proporcionan funcionalidad agregada. Estos se han asociado aún más a las diferentes versiones de ROS y consigo los diferentes paquetes que estos tengan.

*Stacks Manifests*: los *Stacks Manifests (stack.xml)* proporcionan la información de un *stack*, como las dependencias en otros stacks y licencias.

*Messages (msg)*: Los tipos de mensajes definen las estructuras de los mensajes enviados en ROS

*Services (srv)*: Definen la solicitud y la estructura de los datos de respuesta de ROS.

El nivel de computación gráfica maneja los siguientes conceptos básicos:

*Nodos*: Los nodos son procesos que llevan a cabo cálculos y/o tareas, ROS está diseñado para trabajar de una manera modular, por lo que un sistema de control para un robot estaría comprendido de muchos nodos, cada nodo se encargaría de una tarea diferente del robot. Los Nodos más comunes son el publicador (publica datos, información, etc) y el suscriptor (recibe datos, información etc).

*Master*: El *master* permite realizar el registro de los nombres y la búsqueda de paquetes, nodos o stacks. Sin el *master*, los nodos no serían capaces de comunicarse entre sí, ni invocar los servicios. Dentro del *master* podemos encontrar los *parámetros de servicio* que permiten almacenar los datos en un lugar seguro.

*Messages:* Los *Messages* permiten la comunicación entre los nodos, estos tienen una estructura simple, dentro de ellos se definen el tipo de datos que serán intercambiados entre los nodos. Entre estos tipos encontramos enteros, flotantes, booleanos entre otros.

*Topics:* Los *Messages* se enrutan por un sistema de transporte, es decir, un nodo envía un mensaje mediante un publicador a un *topic* determinado. El *topic* es utilizado para identificar el contenido de un *messages*, ya que un nodo que esté interesado en un determinado tipo de dato se suscribirá al tema correspondiente. Pueden existir muchos publicadores y suscriptores a un mismo *topic*, y un nodo puede publicar o suscribirse a muchos *topics*. Esto se puede ver en (Figura 3.1)

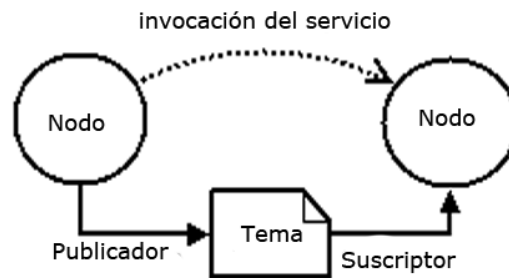


Figura 3.1. Funcionamiento básico de temas y nodos.  
Fuente: Modificado de [19].

*Servicios:* el modelo publicador/suscriptor es un paradigma de comunicación muy flexible, aunque el medio de transporte no es muy apropiado para las interacciones de pregunta respuesta, que a menudo se requieren en un sistema distribuido. La pregunta/respuesta se realiza mediante servicios que son definidos mediante un par de estructuras de mensajes, uno para la pregunta y otro para la respuesta. Un nodo ofrece un servicio con un nombre y un cliente utiliza el servicio enviando un mensaje de solicitud y espera la respuesta. La librería *Client* de ROS presenta generalmente esta interacción para el programador como si fuera una llamada de procedimiento remoto.

*bags:* Las bolsas son un formato para guardar y reproducir datos de mensajes ROS, son un mecanismo importante para el almacenamiento de datos, tales como datos de sensores, que pueden ser difíciles de recoger, pero necesarios para desarrollar y probar algoritmos.

El *master* de ROS actúa como un servicio en este nivel de computación gráfica, él toma la información de *topics* y *services*, y la registra para el uso de nodos. Los nodos se comunican con el *master* reportando la información a la cual están registrados. Los nodos se conectan directamente a otros nodos, como un servidor DNS y de este modo pueden intercambiar mensajes a través de los *topics* disponibles, de este modo un nodo que emite la lectura de un sensor puede enviar los valores de lectura por medio de un *topic* en el cual se publican dichos valores, de tal modo que otro nodo se suscriba al *topic* para obtener la lectura de los sensores. Los nodos que se suscriben a un *topic* pueden preguntar por las conexiones de nodos que publica ese *topic*, y así poder establecer una conexión entre ellos mediante un protocolo.

### 3.1.2 Instalación de ROS

Una versión de ROS puede ser incompatible con otra, de igual forma ciertas versiones de ROS son compatibles únicamente con ciertas versiones de Ubuntu.

Normalmente están referidas por un sobrenombre en vez de por una versión numérica. Las versiones, desde la más actual a la primera versión, son:

- 31/Diciembre/2012 - *Groovy Galapagos*
- 23/Abril/2012 - *Fuerte*
- 30/Agosto/2011 - *Electric Emys*
- 02/Marzo/2011 - *Diamondback*
- 03/Agosto/2010 - *C Turtle*
- 01/Marzo/2010 - *Box Turtle*
- 22/Enero/2010 - *ROS 1.0*

En este caso, se trabajó con la versión 11.10 de Ubuntu, la cual es compatible con ROS en sus versiones *Electric Emys*, *Fuerte* y *Groovy Galapagos*, de las cuales se trabajó sobre la versión de ROS *Electric Emys* (accesible desde <http://ros.org/wiki/electric>).

Para poder instalar ROS es primordial tener instalado Ubuntu de manera previa, es importante tener actualizado el sistema operativo, para lo cual debemos tener abierta una terminal de comandos de Ubuntu, para ello debemos dirigirnos al botón inicio de Ubuntu como se muestra en la Figura 3.2.

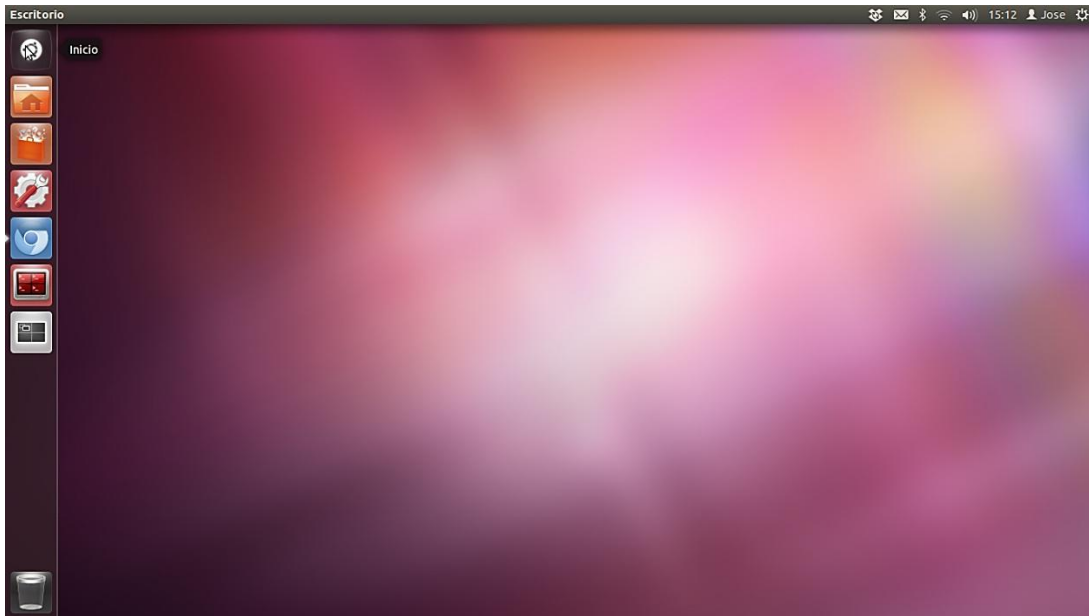


Figura 3.2. Pantalla de inicio de Ubuntu.  
Fuente: Propia.

Una vez abierto el menú inicio en la barra de búsqueda ingresamos la palabra “terminal”, lo cual nos dirigirá al acceso directo del terminal de Ubuntu (Figura 3.3). De aquí en adelante se deberá repetir este procedimiento para la obtención de una nueva terminal.

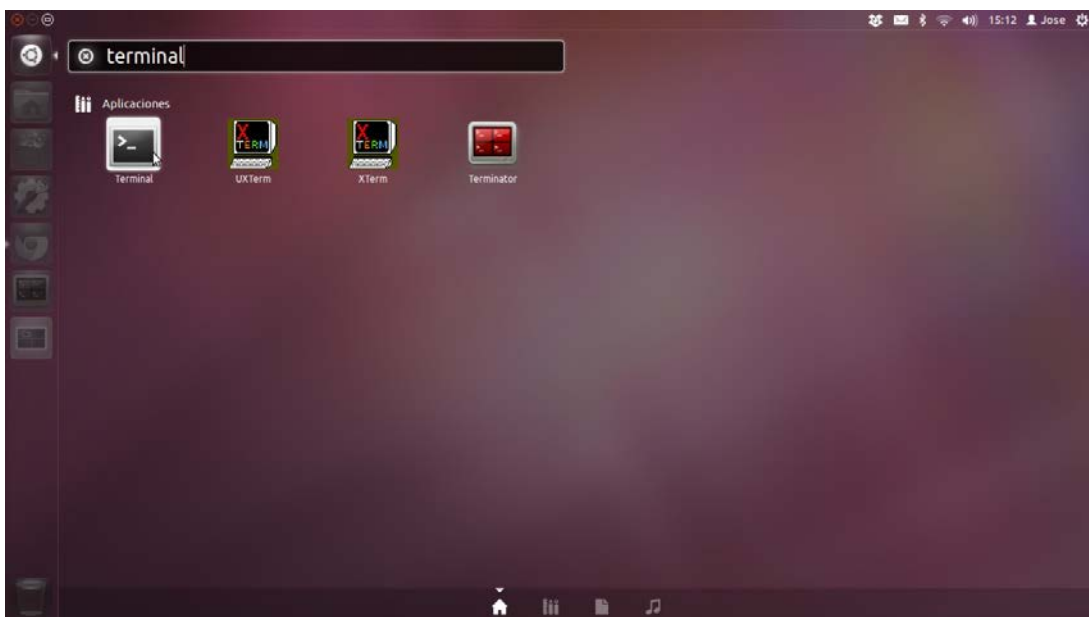


Figura 3.3. Pantalla de búsqueda del terminal.  
Fuente: Propia.



En la terminal abierta se ingresa el siguiente comando para poder actualizar los paquetes de Ubuntu:

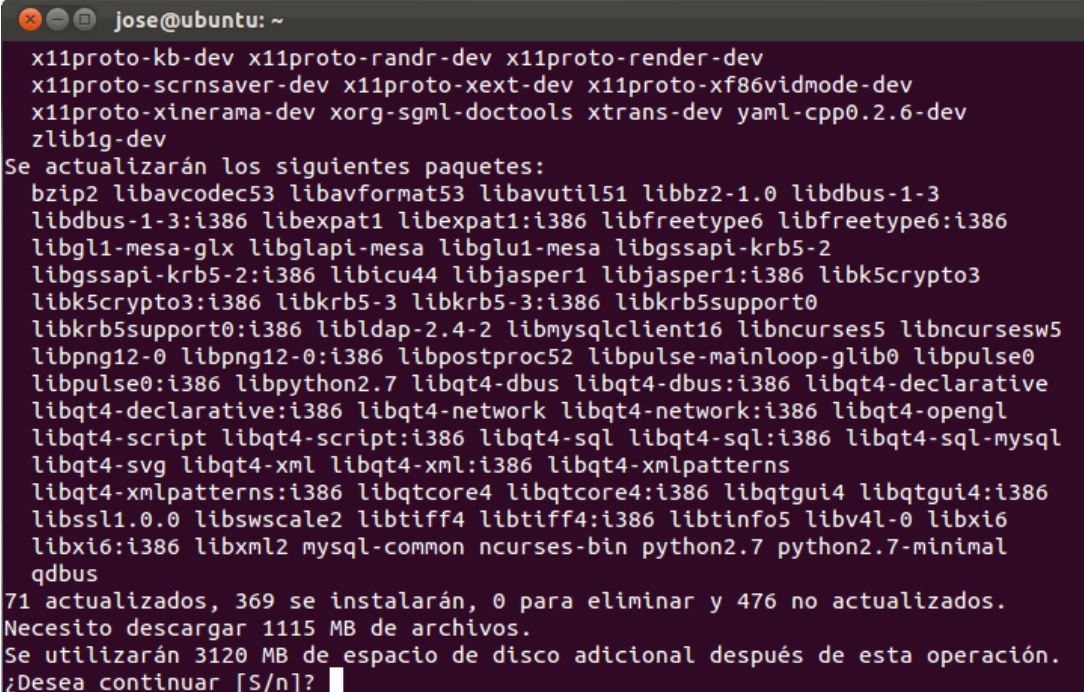
```
sudo apt-get update
```

Una vez ingresado este comando debemos presionar la tecla enter y esperar a que el proceso se complete.

Una vez actualizado el sistema en la misma terminal se prosigue con la instalación de ROS, para ello se necesita ingresar la siguiente línea en dicha terminal:

```
sudo apt-get install ros-electric-desktop-full
```

Una vez iniciada la instalación en la terminal se nos preguntara si deseamos continuar con la instalación para lo cual deberemos teclear “S” y presionar enter, tal y como se presenta en la Figura 3.4.



```
jose@ubuntu: ~
x11proto-kb-dev x11proto-randr-dev x11proto-render-dev
x11proto-scrnsaver-dev x11proto-xext-dev x11proto-xf86vidmode-dev
x11proto-xinerama-dev xorg-sgml-doctools xtrans-dev yaml-cpp0.2.6-dev
zlib1g-dev
Se actualizarán los siguientes paquetes:
bzip2 libavcodec53 libavformat53 libavutil51 libbz2-1.0 libdbus-1-3
libdbus-1-3:i386 libexpat1 libexpat1:i386 libfreetype6 libfreetype6:i386
libgl1-mesa-glx libglapi-mesa libglu1-mesa libgssapi-krb5-2
libgssapi-krb5-2:i386 libicu44 libjasper1 libjasper1:i386 libk5crypto3
libk5crypto3:i386 libkrb5-3 libkrb5-3:i386 libkrb5support0
libkrb5support0:i386 libldap-2.4-2 libmysqlclient16 libncurses5 libncursesw5
libpng12-0 libpng12-0:i386 libpostproc52 libpulse-mainloop-glib0 libpulse0
libpulse0:i386 libpython2.7 libqt4-dbus libqt4-dbus:i386 libqt4-declarative
libqt4-declarative:i386 libqt4-network libqt4-network:i386 libqt4-opengl
libqt4-script libqt4-script:i386 libqt4-sql libqt4-sql:i386 libqt4-sql-mysql
libqt4-svg libqt4-xml libqt4-xml:i386 libqt4-xmlpatterns
libqt4-xmlpatterns:i386 libqtcore4 libqtcore4:i386 libqtgui4 libqtgui4:i386
libssl1.0.0 libswscale2 libtiff4 libtiff4:i386 libtinfo5 libv4l-0 libxi6
libxi6:i386 libxml2 mysql-common ncurses-bin python2.7 python2.7-minimal
qdbus
71 actualizados, 369 se instalarán, 0 para eliminar y 476 no actualizados.
Necesito descargar 1115 MB de archivos.
Se utilizarán 3120 MB de espacio de disco adicional después de esta operación.
¿Desea continuar [S/n]?
```

Figura 3.4. Instalación ROS.  
Fuente: Propia.

Aquí se instala la versión completa de ROS *Electric*, incluyendo las librerías de robots implementados en ROS y simuladores de 2D y 3D.

Es necesario que las variables de entorno de ROS se agreguen automáticamente cada vez que una nueva sesión es iniciada, para ello se

debe ingresar el siguiente comando en la terminal y presionar *enter*:

```
Echo "source /opt/ros/electric/setup.bash" >> ~/.bashrc  
. ~/.bashrc
```

ROS necesita algunas herramientas independientes debido a que *rosinstall* y *rosdep* utilizan con frecuencia líneas de comandos de ROS que se distribuyen por separado. (*rosinstall* permite instalar fácilmente las dependencias al sistema de las fuentes que se vayan a compilar) Para ello corremos la siguiente línea:

```
sudo apt-get install python-rosinstall python-rosdep
```

De este modo la instalación de ROS en su versión *Electric* concluye.

## 3.2 PAQUETES DE ROS UTILIZADOS

### 3.2.1 Gazebo

Gazebo es un simulador multi-robot para entornos de software libre, es capaz de simular una población de robots, sensores y objetos en un mundo tridimensional, simula una gran cantidad de características físicas, la gravedad, colisiones, realimentación de sensores entre otras. Gazebo se encuentra en constante desarrollo con *Robotic Open Source Foundation*, corrigiendo errores y agregando nuevas características. Gazebo está incluido en ROS como un paquete que se puede instalar dentro de las dependencias de ROS [19].

### 3.2.2 Rosserial

*Rosserial* es un paquete que permite la integración de Arduino, microcontroladores o cualquier tipo de dispositivo hardware con conexión serial con ROS, *roserial* consiste en un protocolo general p2p. En este proyecto se utilizaron *roserial\_arduino* (librería cliente) y *roserial\_python* (interfaz PC/Tablet.) [19].

El protocolo *roserial* posee tres paquetes disponibles los cuales son los siguientes:

*rosserial\_arduino*: Este paquete contiene las extensiones necesarias para ejecutar *rosserial\_client* en Arduino.

*rosserial\_client*: Este paquete contiene la implementación del cliente *rosserial*. Diseñado para los microcontroladores que tengan un compilador ANSI C++ y un puerto serial a un computador.

*rosserial\_python*: El paquete contiene la implementación de la conexión del puerto host. Se encarga de la instalación, la publicación y la suscripción de un dispositivo *rosserial* habilitado.

### 3.2.3 URDF

Este paquete hace parte del paquete *Robot Model* que se encuentra en [19] y viene incluido en la instalación completa de ROS, el cual contiene dentro de sí otros paquetes para modelar robots, en los cuales se especifican todas las geometrías y características físicas de un robot. El más usado es URDF (*unified robot description format*), que es un formato XML para el modelamiento de un robot. Este paquete contiene numerosas especificaciones para la descripción de robots y sensores [19], estas especificaciones abarcan: descripción cinemáticas y dinámicas del robot, representación visual del robot, y modelo de colisión del robot. Se asume que el robot está formado por eslabones rígidos, cada uno sería un *link*, unidos unos a otros por articulaciones (*joints*). Un ejemplo simple del uso de un archivo URDF es el modelo simple que se ve en la Figura 3.5, el archivo XML se explica a continuación:

En esta primera sección se inicia la descripción del robot y se le asigna un nombre, en este caso `simple_box`. Se crea el primer eslabón (*link*) llamado "my\_box1". Dentro de la descripción `<inertial>` se describe el origen, masa e inercia de ese *link*, y dentro de la descripción visual se define la geometría y color (se pueden dar más características).

```
<robot name="simple_box">
  <link name="my_box1">
    <inertial>
      <origin xyz="0 0 0" />
      <mass value="1.0" />
      <inertia ixx="1.0"
ixy="0.0" ixz="0.0" iyy="100.0" iyz="0.0" izz="1.0" />
    </inertial>
    <visual>
```

```

    <origin xyz="0 0 0"/>
    <geometry>
      <box size="0.2 0.2 0.6" />
    </geometry>
  </visual>
  <collision>
    <origin xyz="0 0 0"/>
    <geometry>
      <box size="0.2 0.2 0.6" />
    </geometry>
  </collision>
</link>
<gazebo reference="my_box1">
  <material>Gazebo/Blue</material>
</gazebo>

```

En la segunda sección se encuentra la descripción de un segundo eslabón, con una nueva etiqueta, `origin`. Esta permite ubicar el sistema de referencia en `xyz`, y definir los ángulos, en radianes, para el eje de rotación, inclinación y orientación en `r`, `p`, y `y` respectivamente. También está la primera junta (*joint*) del ejemplo, en donde también se le da un nombre (`base_to_right_leg`), se especifica el origen, ubicación deseada y, se especifica el tipo de junta que es, existen seis tipos (*revolute*, *continuous*, *prismatic fija*, *float flotante*, *planar*).

```

<link name="my_box2">
  <inertial>
    <origin xyz="0 0 0" />
    <mass value="1.0" />
    <inertia ixx="1.0"
ixy="0.0" ixz="0.0" iyy="100.0" iyz="0.0" izz="1.0" />
  </inertial>
  <visual>
    <origin xyz="0 0 0"/>
    <geometry>
      <box size="0.2 0.2 0.6" />
    </geometry>
  </visual>
  <collision>
    <origin xyz="0 0 0"/>
    <geometry>
      <box size="0.2 0.2 0.6" />
    </geometry>
  </collision>
</link>
<gazebo reference="my_box2">
  <material>Gazebo/Green</material>
</gazebo>

```

```

<joint name="base_to_right_leg" type="fixed">
  <parent link="my_box1"/>
  <child link="my_box2"/>
  <origin xyz="0 0 0.6"/>
</joint>

```

En esta última parte se tiene otro eslabón el cual tiene características como la inercia (*inertial*), aquí se especifica la masa y la matriz de inercia del eslabón, y se realiza la descripción visual. Dentro de este eslabón también está un ejemplo de la definición de la geometría que colisiona, en este se define el origen de la geometría, y la geometría como tal del objeto a colisionar, por lo general y en la mayoría de los casos se usa la misma geometría usada en la descripción visual del eslabón.

```

<link name="my_box3">
  <inertial>
    <origin xyz="0 0 0" />
    <mass value="1.0" />
    <inertia ixx="1.0"
ixy="0.0" ixz="0.0" iyy="100.0" iyz="0.0" izz="1.0" />
  </inertial>
  <visual>
    <origin xyz="0 0 0"/>
    <geometry>
      <box size="0.2 0.2 0.6" />
    </geometry>
  </visual>
  <collision>
    <origin xyz="0 0 0"/>
    <geometry>
      <box size="0.2 0.2 0.6" />
    </geometry>
  </collision>
</link>
<gazebo reference="my_box3">
  <material>Gazebo/Red</material>
</gazebo>

<joint name="base_to_left_leg" type="fixed">
  <parent link="my_box2"/>
  <child link="my_box3"/>
  <origin xyz="0 0 0.6"/>
</joint>
</robot>

```

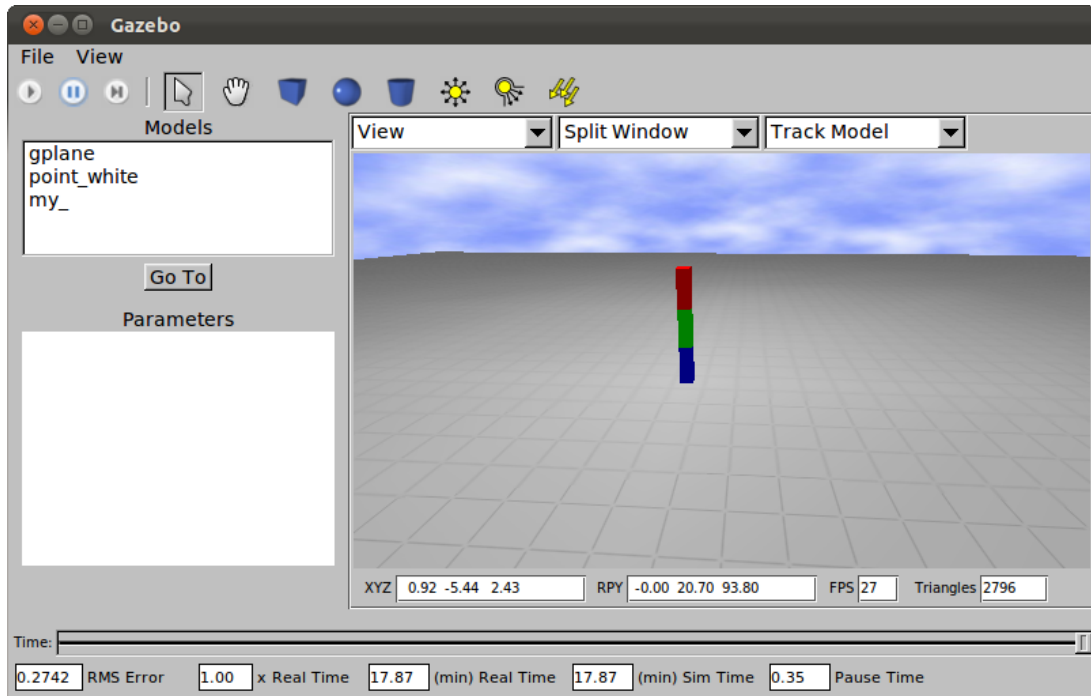


Figura 3.5. Modelo simple de un robot con URDF.  
Fuente: Propia.

### 3.2.4 Shadow Robot

Shadow Robot es una de las empresas de robótica más antiguas en el Reino Unido. En los últimos años se han especializado en el desarrollo de la robótica humanoide, lo que los llevó a crear la mano de un robot llamada “Shadow Hand” (Figura 3.6). En ROS encontramos un paquete que incluye todos los modelos de esta mano así como también los controladores para la manipulación de esta en el entorno virtual [19].



Figura 3.6. Shadow Hand.  
Fuente: [19].

## 3.2.5 Instalación e inicialización de los paquetes de ROS utilizados

### 3.2.5.1 Instalacion e Inicialización de Gazebo

El simulador Gazebo viene inmerso en el entorno de ROS, es decir que al instalar la versión completa de ROS *Electric* este instala de forma automática el simulador, sin embargo es necesario correr las siguientes líneas para un correcto funcionamiento del simulador.

```
sudo apt-get install ros-electric-simulator-gazebo
rosmake simulator_gazebo
```

Para poder hacer uso del entorno gráfico de Gazebo se debe iniciar el mismo, haciendo uso de una terminal nueva introducimos los siguientes comandos:

```
roslaunch gazebo_worlds empty_world.launch
```

Una vez ingresada la anterior línea proseguimos a presionar la tecla *enter*, con lo cual se abrirá el entorno como es presentado en la Figura 3.7.

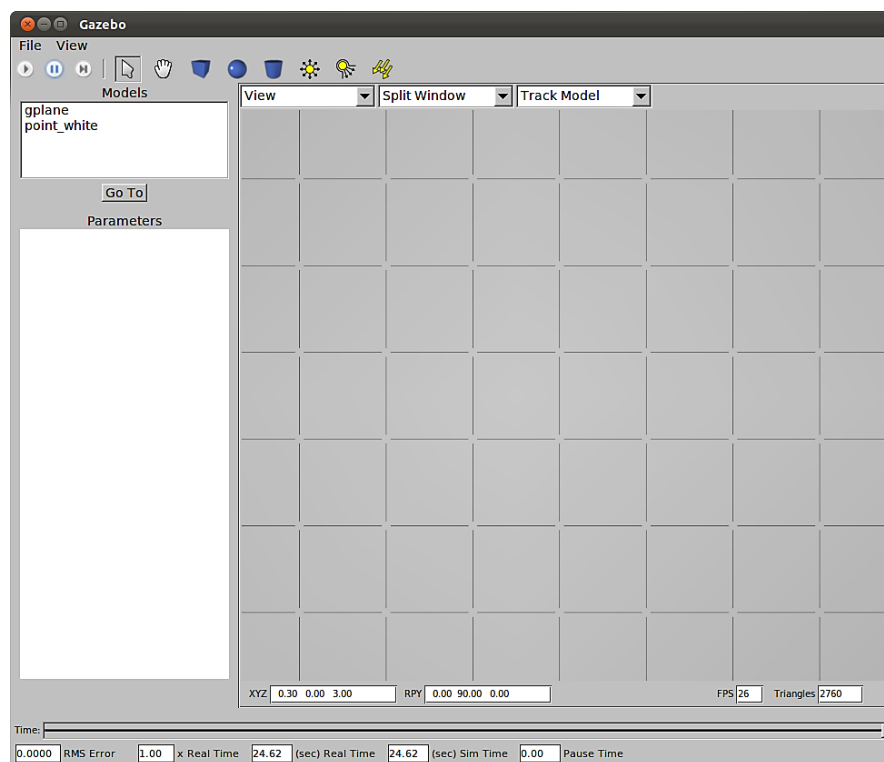


Figura 3.7. Entorno gráfico de Gazebo.

Fuente: Propia.

### 3.2.5.2 Instalación de `rosserial_arduino`

Para instalar el paquete de `rosserial_arduino` se debe abrir un terminal vacío e ingresar el siguiente comando:

```
sudo apt-get install ros-electric-rosserial
```

Desde la anterior línea de comando se instala el repositorio completo del `rosserial`. Finalmente lo que se tiene que hacer es copiar la carpeta `rosserial_arduino/libraries` en el `sketchbook` de Arduino, para ello se corren las siguientes líneas:

```
roscd rosserial_arduino/libraries  
cp r ros_lib <sketchbook>/libraries/ros_lib
```

Una vez hecho esto, chequearemos que la librería se haya añadido al IDE Arduino. Abrimos Arduino en el menú principal vamos a File -> Examples y deberá aparecer `ros_lib` en la lista. Si no se encuentra ahí revisar en File -> Sketchbook, lo indicado anteriormente se encuentra explicado en la Figura 3.8.

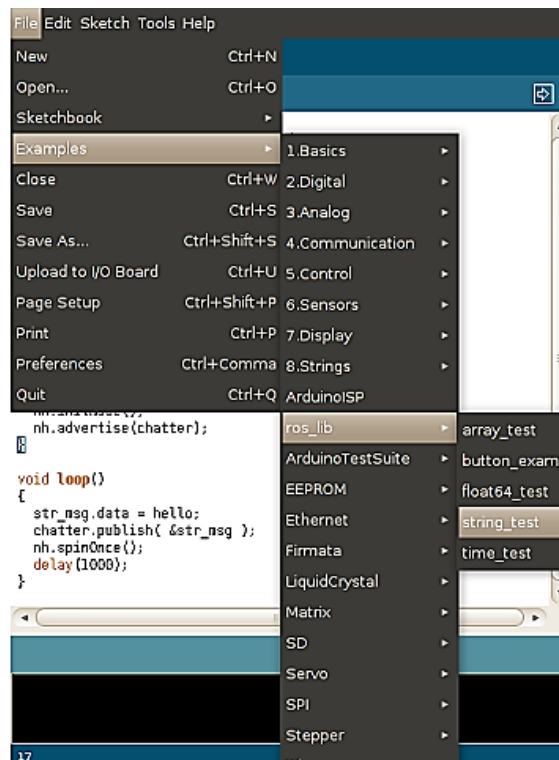


Figura 3.8. IDE de Arduino.

Fuente: Propia.



### 3.2.5.3 Instalación de Shadow Hand

Para la Instalación del paquete *Shadow Hand* el cual es de vital importancia en el desarrollo del proyecto se debe crear inicialmente un espacio de trabajo para poder instalar una versión de *Shadow Hand* diferente a la que se trabaja en la versión *Electric*, para esto se debe empezar por crear el espacio de trabajo al que llamaremos *Shadow*, abriendo un nuevo terminal e ingresando las siguientes líneas de comando:

```
mkdir ~/shadow
source /opt/ros/electric/setup.bash
rosws init ~/shadow /opt/ros/electric
```

una vez creado el espacio de trabajo se deben correr las siguientes líneas para dar por terminada la creación del nuevo espacio de trabajo y exportar la ubicación actual:

```
rosws init ~/shadow /opt/ros/electric
```

Una vez creado el espacio de trabajo debemos crear un archivo de texto que se llamara *shadow\_robot.rosinstall* que incluya la siguiente información y se guarda en la ubicación */tmp/shadow\_robot.rosinstall*.

```
- bzh:
  uri: 'lp:sr-ros-interface'
  local-name: shadow_robot
- bzh:
  uri: 'lp:sr-ros-interface-ethernet'
  local-name: shadow_robot_ethernet
- bzh:
  uri: 'lp:sr-config'
  local-name: sr_config
- bzh:
  uri: 'lp:sr-visualization'
  local-name: sr_visualization
- bzh:
  uri: 'lp:sr-teleop'
  local-name: sr_teleop
```

Una vez creado el archivo de texto se debe agregar el archivo *rossinstall* al espacio de trabajo creado anteriormente, para ello en un terminal nuevo se deben correr las siguientes líneas de comando:

```
source ~/shadow/setup.bash
```

```
rosws merge /tmp/shadow_robot.rosinstall
rosws up
```

En este punto ya se tiene un espacio de trabajo con los *stacks* necesarios para correr *Shadow Hand*, sin embargo deberá compilarse el paquete y algunos otros paquetes necesarios para el correcto funcionamiento como se muestra a continuación:

```
source ~/shadow/setup.bash
export ROS_PACKAGE_PATH=${ROS_PACKAGE_PATH}:'`pwd`'
sudo apt-get install ros-electric-pr2-simulator
sudo apt-get install ros-electric-pr2-controllers
sudo apt-get install ros-electric-arm-navigation
sudo apt-get install ros-electric-object-manipulation
rosmake pr2_gazebo_plugin
rosmake sr_hand
rosmake sr_description
rosmake sr_tactile_sensors
roscdep install sr_utilities sr_hardware_interface sr_description
sr_mechanism_model sr_movements sr_tactile_sensors sr_hand
sr_mechanism_controllers sr_robot_msgs sr_move_arm sr_kinematics
sr_gazebo_plugins sr_example
rosmake shadow_robot
```

Para poder ejecutar la *Shadow Hand* se debe iniciar primero el entorno gráfico de Gazebo, para ello debemos ingresar el siguiente comando en un terminal nuevo.

```
roslaunch gazebo_worlds empty_world.launch
```

Una vez cargado el entorno gráfico se debe correr la siguiente línea en una terminal nueva para cargar *Shadow Hand* como se ve en la Figura 3.9.

```
roslaunch sr_hand gazebo_arm_and_hand.launch
```

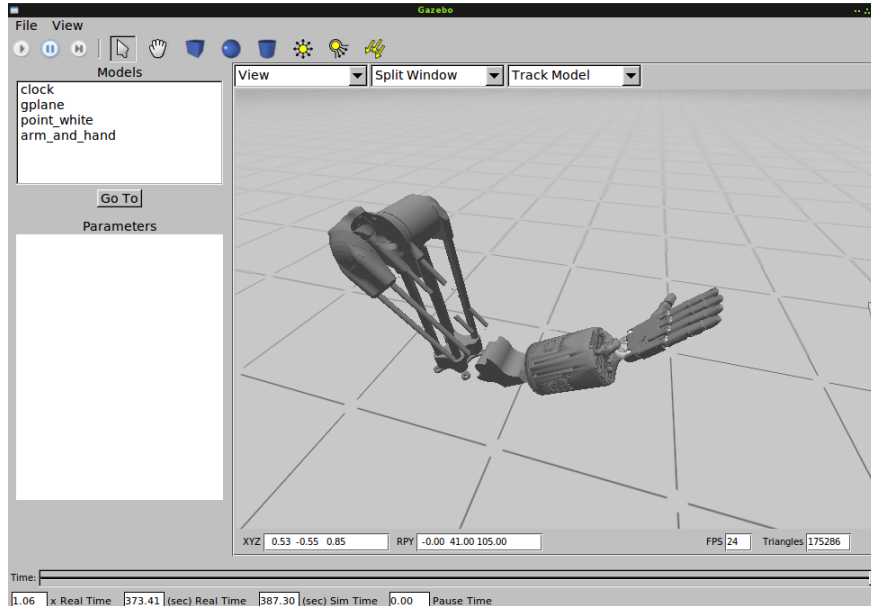
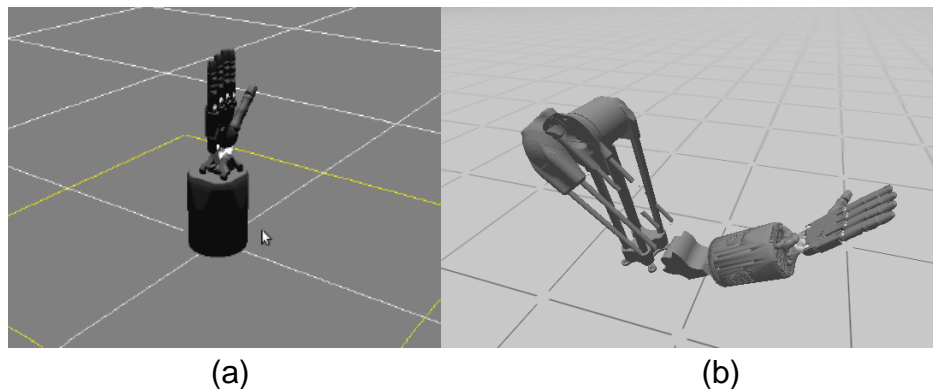


Figura 3.9. Shadow Hand en Gazebo.  
Fuente: Propia.

### 3.3 RECONOCIMIENTO DE SHADOW HAND

La implementación de Shadow Hand dentro de ROS cuenta con dos opciones de la misma, una en la que solo se encuentra la mano con antebrazo (Figura 3.10 (a)), algo inestable para los movimientos realizados con VirtualTouch por todas las fuerzas e inercias que maneja Gazebo, y la mano completa con brazo y un soporte adicional (Figura 3.10 (b)), presentando una mayor estabilidad ante la gravedad, y ante cualquier movimiento brusco, razón por la cual se decidió utilizar esta opción para el desarrollo de este proyecto.



(a) (b)  
Figura 3.10. Shadow Robot en sus dos versiones.  
Fuente: Propia.

Para este proyecto se tiene en cuenta solamente las articulaciones que hacen parte de la mano ya que son las articulaciones que finalmente se moverán con

el guante, Shadow Hand está compuesta por 16 articulaciones, distribuidas de la siguiente manera:

- *Dedo índice:* Está compuesto por tres articulaciones principales, falange proximal (FFJ3), falange media (FFJ2) y falange distal (FFJ1).
- *Dedo Corazón:* Está compuesto por tres articulaciones, falange proximal (MFJ3), falange media (MFJ2) y falange distal (MFJ1).
- *Dedo Anular:* Está compuesto por tres articulaciones principales, falange proximal (RFJ3), falange media (RFJ2) y falange distal (RFJ1).
- *Dedo Meñique:* Está compuesto por cuatro articulaciones principales, falange proximal (LFJ3), falange media (LFJ2), falange distal (LFJ1) y una cuarta articulación ubicada en el extremo de la palma de la mano la cual le da cierta elasticidad y facilidad al movimiento de la mano (LFJ5).
- *Dedo Pulgar:* Está compuesto por tres articulaciones principales, la base (THJ4), falange media (THJ2) y falange distal (THJ1)

Para mayor referencia de las falanges proximal, media y distal, ver Figura 3.11.



Figura 3.11. Anatomía de la mano.  
Fuente: [20].

Además de las falanges anteriormente mencionadas, los dedos índice, corazón, anular y meñique cuentan con un movimiento horizontal extra ubicado en las articulaciones FFJ4, MFJ4, RFJ4 y LFJ4 respectivamente, que permite a los dedos la libertad de hacer una flexión lateral de la falange proximal.

El dedo pulgar posee algunos movimientos extra de rotación en su propio eje asignados en las articulaciones THJ5 y THJ3.

Cada una de las articulaciones tiene asignado un *topic* en ROS, por el cual se le puede publicar un valor (ángulos en unidades de radianes). Estos pasan por un controlador PID (desarrollado por Shadow Hand), permitiendo y simulando así el movimiento real en la pantalla, es decir, si un nodo de ROS está emitiendo un valor en radianes por la lectura de un sensor o sencillamente por una operación interna, este valor puede ser publicado en el *topic* de alguna de las articulaciones, de este modo la articulación generara el movimiento simulado en el entorno gráfico.

Para acceder a los *topics* que dispone *Shadow Hand* se realiza lo siguiente:

En una terminal nueva se corre la siguiente línea de comando que abre el entorno de Gazebo con un mundo vacío:

```
roslaunch gazebo_worlds empty_world.launch
```

Una vez abierto Gazebo, se carga la mano por medio de la siguiente línea de comando en una terminal nueva:

```
roslaunch sr_hand gazebo_hand.launch
```

Una vez cargada la mano en la interfaz de Gazebo, en una terminal nueva se teclea la siguiente línea de comando para obtener el listado de *topics* disponible para la *Shadow Hand* y Gazebo.

```
rostopic list
```

De esta manera se pueden localizar los *topics* disponibles para publicar y suscribir nodos en la *Shadow Hand*. Se buscan aquellos relacionados con la posición de las diferentes articulaciones mencionadas anteriormente. Los *topics* que reciben los ángulos para generar el movimiento de los dedos de la mano son:

```
/sh_ffj0_mixed_position_velocity_controller/state  
/sh_ffj3_mixed_position_velocity_controller/state  
/sh_ffj4_mixed_position_velocity_controller/state  
/sh_lfj0_mixed_position_velocity_controller/state  
/sh_lfj3_mixed_position_velocity_controller/state  
/sh_lfj4_mixed_position_velocity_controller/state  
/sh_lfj5_mixed_position_velocity_controller/state  
/sh_mfj0_mixed_position_velocity_controller/state  
/sh_mfj3_mixed_position_velocity_controller/state  
/sh_mfj4_mixed_position_velocity_controller/state  
/sh_rfj0_mixed_position_velocity_controller/state  
/sh_rfj3_mixed_position_velocity_controller/state  
/sh_rfj4_mixed_position_velocity_controller/state  
/sh_thj1_mixed_position_velocity_controller/state
```

```
/sh_thj2_mixed_position_velocity_controller/state  
/sh_thj3_mixed_position_velocity_controller/state  
/sh_thj4_mixed_position_velocity_controller/state  
/sh_thj5_mixed_position_velocity_controller/state  
/sh_wrj1_mixed_position_velocity_controller/state  
/sh_wrj2_mixed_position_velocity_controller/state
```

Como es posible ver en la anterior lista no se encuentran los *topics* asignados a las falanges media y distal de los dedos, y esto es porque las articulaciones de dichas falanges no poseen un movimiento independiente, y para darle mayor naturalidad al movimiento de la mano se han asignado *topics* como “/sh\_ffj0\_mixed\_position\_velocity\_controller/state”. Todos los *topics* con *joint* cero (J0) tienen una estructura compartida con los *joints* J1 y J2, es decir,  $J0 = J1+J2$  de tal modo si a FFJ0 le envío un valor de  $2*\pi/3$ , las articulaciones FFJ1 y FFJ2 recibirán  $\pi/2$ , respectivamente, esto con el fin de simular el movimiento de la falange distal de cada dedo, ya que naturalmente la mano humana en muy pocos casos genera el movimiento independiente de esta falange. Por este motivo se simula el movimiento de la falange distal como la mitad del ángulo generado por la falange proximal.

Cabe anotar que la *Shadow Hand* posee dos movimientos en la muñeca de la mano designados en las articulaciones WRJ1 y WRJ2 los cuales no serán utilizados en este caso.

### 3.4 QT

Qt es una biblioteca multiplataforma utilizada principalmente para el desarrollo de aplicaciones con una interfaz gráfica de usuario, utilizando C++ o QML, un CSS y JavaScript como lenguaje. Qt Creator es el entorno de desarrollo, donde se pueden editar visualmente las interfaces diseñadas. Todas las herramientas de Qt son desarrolladas bajo el concepto *open-source* [21].

Qt incluye las herramientas necesarias para el diseño de aplicaciones de una forma rápida y sencilla, la biblioteca incluye un conjunto de *widgets* encargados de proporcionar las funciones estándar de una interfaz o GUI. También cuenta con una alternativa para la comunicación entre-objetos denominada *signals* y slots, además de proporcionar un modelo de eventos convencional para la captura de las pulsaciones del ratón, teclas u otras entradas del usuario haciendo posible la creación de las aplicaciones multiplataforma mediante funciones estándar.

Con QT se diseñó la interfaz de usuario para la implementación con VTK y VCollide, bajo el sistema operativo Windows.

### 3.5 VTK

VTK (*Visualization Toolkit*) es un conjunto de librerías desarrollado por Kitware, permite la visualización de geometrías 3D, soportando una amplia variedad de algoritmos de visualización y modelado. Es una herramienta *Open Source*, lo que la ha impulsado en su difusión extendiendo su aplicación a diversos campos donde se emplean objetos 3D. Las funciones que integran VTK se basan en los principios de orientación a objetos, conformando un complejo sistema jerárquico de filtros y fuentes que dotan de gran flexibilidad a todo el sistema [22].

EL modelo VTK está basada en el paradigma del flujo de datos adoptado por la mayoría de los sistemas comerciales. Según este paradigma los diferentes módulos son conectados formando una red. La ejecución de la red de visualización es controlada en respuesta a las demandas de datos (conducción por demanda) o en respuesta a una entrada del usuario (conducción por eventos). La gran ventaja de este sistema es que es flexible, y puede ser rápidamente adaptado a diferentes tipos de datos o a nuevas implementaciones algorítmicas. La parte de visualización es la encargada de generar los datos que serán presentados por el modelo gráfico. VTK utiliza un flujo de datos aproximado para transformar información en datos gráficos.

### 3.6 VCollide

V-Collide es una librería de detección de colisiones desarrollada por el grupo GAMMA (*Geometric algorithm for modeling, motion and animation*) de la Universidad de Carolina del Norte. Realiza una eficiente y exacta detección de colisiones entre modelos poligonales, diseñada para operar en ambientes que contienen un gran número de objetos geométricos formados con mallas de triángulos. La arquitectura de detección de colisiones se basa en tres etapas:

- *Test N-body*: Encuentra parejas de objetos que posiblemente colisionen.
- *Test jerárquico*: Encuentra parejas de triángulos que posiblemente intersecten entre sí, utilizando el árbol jerárquico de cajas orientadas (OBB).
- *Test exacto*: Verifica si las parejas de triángulos realmente colisionan.

Las dos últimas etapas se toman de una librería independiente llamada RAPID.

## 4. DESARROLLO DEL SOFTWARE PARA VIRTUALTOUCH

En el desarrollo de VirtualTouch además de un componente hardware posee un componente software, el cual permite el funcionamiento del dispositivo háptico. Para ello se realizó una implementación software en base al sistema operativo para robots ROS con la cual se alcanzó hasta el nivel de detección de colisiones con objetos virtuales. Con el fin de satisfacer la posibilidad de lograr la deformación de objetos 3d con el uso de VirtualTouch se implementó un software desarrollado en Microsoft Visual Studio 2008 con la ayuda de las librería VTK y VCollide y una interfaz de usuario diseñada en Qt.

Para el correcto desarrollo de las herramientas software se planteó inicialmente la caracterización de los sensores con el fin de obtener las ecuaciones de funcionamiento de cada uno de los *flex sensor* posteriormente se realizó el desarrollo del software en ROS y finalmente el desarrollo con Qt/VTK.

### 4.1 CARACTERIZACIÓN DE LOS FLEX SENSOR

Como se mencionó anteriormente los *flex sensors* consisten en sí de varios sensores que cambian su resistencia en función de la cantidad de curvatura aplicada.

La lectura de los *flex sensors* en la implementación final será mediante una Arduino Mega 2560, para ello se realizó un programa en Arduino que leyera los valores del *flex sensor*. La tarjeta Arduino realmente recibe un valor de voltaje el cual es convertido a bits (0 y 1023), la relación entre voltaje, resistencia y bits es que entre mayor sea la resistencia, mayor voltaje, se transformara a un valor menor en bits. Es decir, en el caso experimental de presentar una curvatura en el sensor de 0° (11.04 KΩ, medida con un multímetro), el voltaje de salida será 2.6v traducido a bits por Arduino tiene un valor promedio de 529, y en el caso de presentar una curvatura de 180° en el sensor, el valor en Arduino será en promedio 386. En la Tabla 4.1 se muestran los resultados obtenidos experimentalmente para 8 grados diferentes, entendiendo así el funcionamiento de los *flex sensor* junto con el circuito implementado y la lectura realizada por Arduino.

Una vez entendido esto, se hizo necesario realizar otra prueba que permitiera ver con mayor precisión el comportamiento del *flex sensor* en el rango de utilización del guante que es de 0° a 90°, debido a que ese es el rango aproximado del movimiento de los dedos humanos. Se realizó la medición de



0° a 90° en pasos de 5 grados como se muestra en la Tabla 4.2, y para una mejor visualización del comportamiento de los datos obtenidos se graficaron como se observa en la Figura.4.1.

N° de muestras	Angulos (resistencia)							
	0° (11,04 KΩ)	30° (12,22 KΩ)	45° (12,70 KΩ)	60° (13,60 KΩ)	90° (14,95 KΩ)	120° (16,30 KΩ)	150° (18,80 KΩ)	180° (20,10 KΩ)
1	529	524	491	478	445	420	405	387
2	529	492	492	478	444	420	402	386
3	530	504	491	478	444	419	404	388
4	530	507	492	479	445	421	403	387
5	529	507	491	478	447	420	405	387
6	529	506	491	479	447	418	404	386
7	530	507	491	478	447	419	404	388
8	529	507	491	479	446	419	405	387
9	528	507	492	478	446	421	403	388
10	530	507	491	478	445	420	404	387
11	529	507	491	478	447	420	403	388
12	530	508	491	479	446	420	401	385
13	529	507	491	479	446	420	402	386
14	530	507	491	478	445	420	404	386
15	529	507	492	478	447	421	403	386
<b>PROMEDIO</b>	529	507	491	478	446	420	403	386

Tabla 4.1. Medición de los *flex sensor* (resistencia – bits).

Ángulos N° muestra	0°	5°	10°	15°	20°	25°	30°	35°	40°	45°	50°	55°	60°	65°	70°	75°	80°	85°	90°
1	398	391	383	374	365	356	348	339	331	323	320	311	306	297	289	282	275	268	260
2	398	390	383	374	365	356	349	339	331	323	319	312	306	297	289	281	275	268	261
3	398	390	383	375	365	357	349	339	332	324	319	312	306	297	289	282	275	268	261
4	398	390	383	374	365	356	349	339	331	324	320	312	306	297	289	282	275	269	261
5	398	390	383	374	365	356	349	339	331	323	320	312	307	297	289	282	275	268	260
6	398	390	383	374	364	356	349	339	331	324	320	312	307	298	289	282	275	269	260
7	398	390	383	374	365	356	349	339	332	324	320	312	306	298	289	282	275	269	262
8	398	391	383	374	366	357	349	339	331	323	320	312	307	298	289	282	276	269	260
9	398	391	383	374	365	357	349	339	331	325	320	312	306	298	290	282	275	269	261
10	398	390	383	374	365	357	349	340	332	324	320	312	307	298	290	282	276	268	261
11	399	391	383	374	364	357	349	340	332	324	320	313	307	298	290	282	276	268	261
12	399	391	383	374	365	357	349	339	332	325	320	312	306	298	290	283	276	268	261
13	398	391	383	374	365	357	350	340	332	324	321	313	307	298	290	283	275	269	261
14	398	390	384	375	365	356	350	340	332	324	320	313	307	297	289	282	276	269	262
15	398	391	383	374	365	356	349	340	333	325	321	313	306	298	289	283	276	270	262
16	398	391	383	375	365	356	349	340	332	325	321	313	307	298	290	283	276	270	262
17	398	391	383	375	365	357	349	340	332	325	321	312	307	298	290	284	276	269	262
18	399	391	383	375	365	357	349	340	332	325	321	313	307	298	290	283	275	270	262
19	398	391	383	374	365	357	349	340	332	325	321	313	307	298	290	282	276	270	262
20	398	390	383	375	365	356	349	340	332	326	321	313	307	298	290	283	276	270	262
21	398	391	383	374	365	357	349	340	332	325	321	313	308	299	290	283	276	270	262
22	399	391	384	376	365	357	349	340	332	325	321	313	308	299	290	283	277	270	262
23	398	391	384	374	366	357	350	340	333	324	320	313	308	299	291	284	276	270	263
<b>Promedio</b>	<b>398,1</b>	<b>390,6</b>	<b>383,1</b>	<b>374,3</b>	<b>365</b>	<b>356,5</b>	<b>349,0</b>	<b>339,6</b>	<b>331,8</b>	<b>324,3</b>	<b>320,3</b>	<b>312,4</b>	<b>306,7</b>	<b>297,8</b>	<b>289,6</b>	<b>282,4</b>	<b>275,6</b>	<b>269</b>	<b>261,3</b>

Tabla 4.2. Toma de datos de *flex sensor* – variación en rango de 0° a 90°.

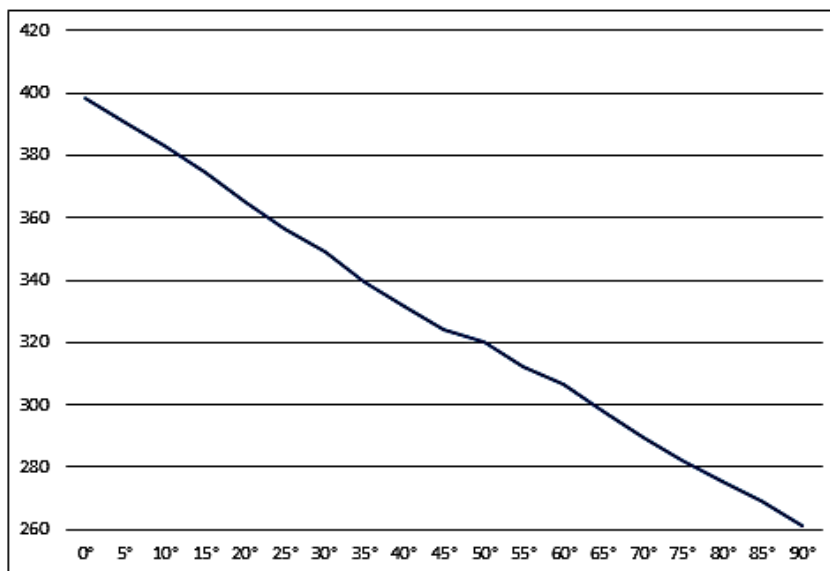


Figura 4.1. Gráfica de datos obtenidos por el *flex sensor* en la variación de 0° a 90°.  
Fuente: Propia.

Al ver que el comportamiento en el rango de utilización de los *flex sensor* se podía aproximar a un comportamiento lineal, se realizó entonces la medición de los *flex sensor* implementados ya en el guante, es decir, realizar la medición dedo por dedo, y haciendo una toma de 0° y 90° por dedo, de donde se obtienen los valores de la Tabla 4.3.

Dedo	Índice		Corazón		Anular		Meñique		Pulgar	
	0	90	0	90	0	90	0	90	0	90
1	449	354	378	222	480	403	449	378	482	420
2	450	356	371	221	482	403	450	378	484	418
3	450	355	368	221	480	405	448	379	482	421
4	449	356	365	219	480	405	450	380	481	420
5	449	356	368	223	480	405	448	380	482	421
6	459	356	380	223	481	404	448	380	481	419
7	451	359	370	223	480	405	451	378	483	421
8	449	356	370	224	480	405	449	380	480	419
9	450	357	377	224	481	406	448	381	481	421
10	448	355	375	224	481	404	449	379	485	422
11	450	357	376	223	482	405	449	380	479	419
12	449	355	380	225	480	407	448	380	481	419
13	451	357	382	223	480	404	449	381	480	420
14	450	357	281	225	480	404	450	380	479	421
15	449	358	381	224	479	405	452	381	479	422
16	449	356	381	225	481	405	448	381	481	423
17	450	357	381	225	481	405	451	380	480	422
18	451	356	380	227	481	405	449	381	481	425
19	449	357	380	224	480	405	450	380	482	421
Promedio	450,11	356,32	370,74	223,42	480,47	404,74	449,26	379,84	481,21	420,74

Tabla 4.3. Medición *flex sensor* para cada dedo.

Finalmente con estos datos se determinó la ecuación lineal respectiva para cada uno de los dedos gracias a los datos obtenidos anteriormente, (se tuvo en cuenta que el controlador implementado dentro de *Shadow Hand* recibe los valores de los ángulos de cada falange de la mano en radianes). Posteriormente estas ecuaciones se implementan en el programa de Arduino para que entregue a ROS valores de ángulos en radianes. A continuación se muestran las ecuaciones finales obtenidas:

Índice:

$$x_i = \frac{450,1 - Y_i}{65,52}$$

Corazón:

$$x_c = \frac{370,7 - Y_c}{98,2}$$

Anular:

$$x_a = \frac{480,47 - Y_a}{50,49}$$

Meñique:

$$x_m = \frac{449,26 - Y_m}{46,28}$$

Pulgar:

$$x_p = \frac{481,21 - Y_p}{40,31}$$

Donde  $Y_i$ ,  $Y_c$ ,  $Y_a$ ,  $Y_m$ , y  $Y_p$ , son los valores sensados por Arduino y  $X_i$ ,  $X_c$ ,  $X_a$ ,  $X_m$ ,  $X_p$ , es el valor en radianes que será entregado a Gazebo para el dedo índice, corazón, anular, meñique y pulgar respectivamente.

## 4.2 DESARROLLO DEL SOFTWARE EN ROS

El desarrollo de la herramienta software en ROS está compuesto por cuatro partes fundamentales, la adquisición de los datos que entregan los sensores por medio de la tarjeta Arduino, el diseño de los suscriptores y publicadores entre la adquisición de datos y *Shadow Hand*, el sistema de detección de colisiones y su respectiva realimentación con el hardware de VirtualTouch y

finalmente la adecuación del entorno gráfico de la herramienta software desarrollada.

#### 4.2.1 Adquisición de datos

Para la adquisición de datos se utilizó la tarjeta Arduino mega 2560. Luego de tener la caracterización de los sensores como se explicó anteriormente, se pudo realizar la implementación completa de los sensores, es decir hacer que Arduino lea los valores que envían los sensores para posteriormente ser enviados a la *Shadow Hand* en Gazebo. Cabe recordar que para la programación de Arduino dentro de ROS se utiliza el paquete *rosserial\_arduino*. El código implementado en Arduino se muestra a continuación:

En la primera parte se declaran las librerías a usar incluyendo la de ROS, y se crea un nodo de ROS llamado en este caso *nh* en la línea 4.

```
#include <ros.h>
#include <indice.h>
#include <std_msgs/Float64.h>
```

```
ros::NodeHandle nh;
```

Luego se declaran los mensajes que enviará Arduino al nodo de ROS y se crean los respectivos *topics* para cada dedo, por ejemplo para el dedo índice se crea un tipo de mensaje Float64 en la primera línea y en la sexta línea se publica en el *topic* índice el mensaje creado anteriormente.

```
std_msgs::Float64 str_msgind;
std_msgs::Float64 str_msgcor;
std_msgs::Float64 str_msganu;
std_msgs::Float64 str_msgmen;
std_msgs::Float64 str_msgpul;
ros::Publisher indice ("indice", &str_msgind);
ros::Publisher corazon("corazon", &str_msgcor);
ros::Publisher anular("anular", &str_msganu);
ros::Publisher menique("menique", &str_msgmen);
ros::Publisher pulgar("pulgar", &str_msgpul);
```

En esta pequeña sección se establecen los puertos de entrada analógica del 0 al 4 para cada dedo.

```
int findice=1;
int fcorazon=2;
```

```
int fanular=3;
int fmenique=4;
int fpulgar=0;
```

Una vez inicializados los puertos se declara unas variables vind, vcor, vanu, vmen y vpul como valores enteros y las variables flotantes para almacenar los valores en radianes que entrega cada sensor.

```
int vind = 0;//valor que sensa el flex sensor valor en bits
int vcor = 0;//valor que sensa el flex sensor valor en bits
int vanu = 0;//valor que sensa el flex sensor valor en bits
int vmen = 0;//valor que sensa el flex sensor valor en bits
int vpul = 0;//valor que sensa el flex sensor valor en bits

float radind;
float radcor;
float radanu;
float radmen;
float radpul;
```

Una vez declaradas las variables pertinentes, se inicializan los puertos de los nodos de ROS que se van a utilizar, se creó uno por cada dedo.

```
void setup(){
  nh.initNode();
  nh.advertise(indice);
  nh.advertise(corazon);
  nh.advertise(anular);
  nh.advertise(menique);
  nh.advertise(pulgar);
}
```

Finalmente la lectura de los sensores se hace por medio del *analogRead* de Arduino en las variables llamadas vind, vcor, vanu, vmen o vpul según el dedo. Se almacena dicho valor que se encuentra en bits (0-1023) y gracias a la caracterización de cada sensor en cada uno de los dedos se llegó a las respectivas ecuaciones para convertir dicho valor en un ángulo en radianes. Una vez convertidos dichos valores se publican en los *topics* respectivos, quienes posteriormente serán suscritos al nodo de cada dedo y publicados en los *topics* de cada articulación de *Shadow Hand* en Gazebo.

```
void loop(){

  vind =analogRead(findice);
  radind = ((450.1052-(vind))/(62.5263));
```

```

str_msgind.data = radind;
indice.publish( &str_msgind);

vcor =analogRead(fcorazon);
radcor = ((370.7368-(vcor))/(98.2105));
str_msgcor.data = radcor;
corazon.publish( &str_msgcor );

vanu =analogRead(fanular);
radanu = ((480.4736-(vanu))/(50.4912));
str_msganu.data = radanu;
anular.publish( &str_msganu );

vmen =analogRead(fmenique);
radmen = ((449.2631-(vmen))/(46.2807));
str_msgmen.data = radmen;
menique.publish( &str_msgmen );

vpul =analogRead(fpulgar);
radpul = ((481.2105-(vpul))/(40.3157));
str_msgpul.data = radpul;
pulgar.publish(&str_msgpul);

nh.spinOnce();
delay(100);
}

```

Una vez se tiene el código en la tarjeta Arduino, se inicia la lectura de los datos. Para ello se inicia en una terminal de Ubuntu el núcleo de ROS y luego en otra terminal se hace el llamado a *rosserial\_python* que directamente creará conexión con el puerto USB que esté conectado. Las líneas que se deben correr respectivamente son:

```

roscore
roslaunch rosserial_python serial_node.py /dev/ttyACM0.

```

En este punto vale aclarar que se debe fijar el nombre del puerto al cual está conectada la Arduino (esto se puede ver claramente en el IDE de Arduino, Herramientas y Puerto Serial), ya que el último trozo de línea dependerá exclusivamente de esto. Por ejemplo en caso de estar conectada la tarjeta al puerto `ttys0`, la segunda línea a correr será:

```

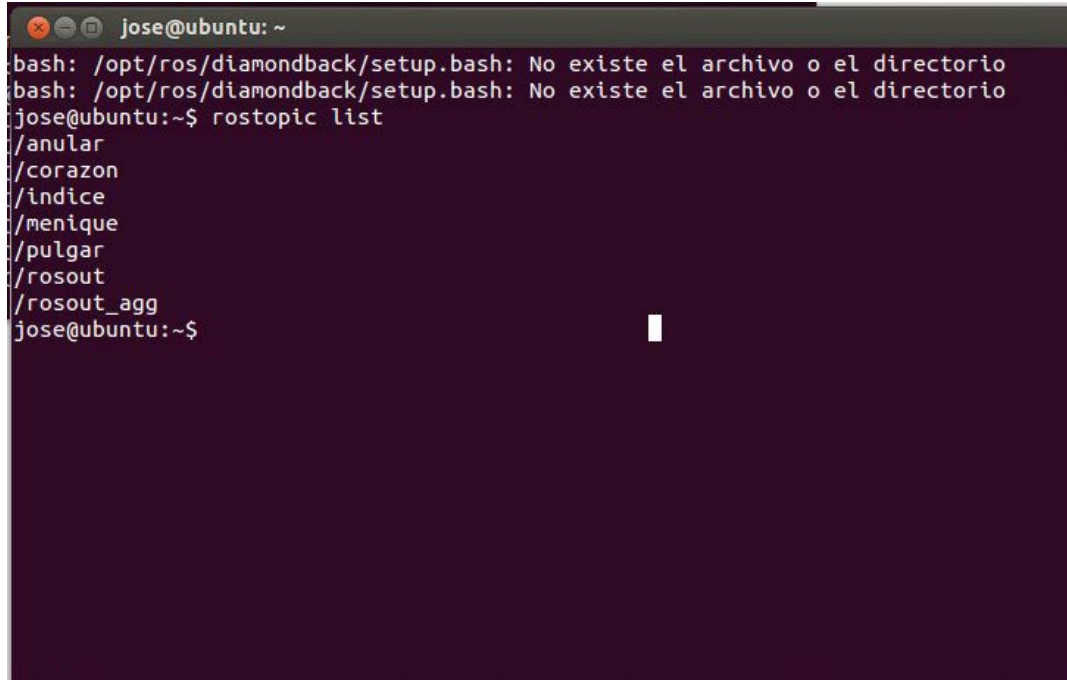
roslaunch rosserial_python serial_node.py /dev/ttyS0.

```

ROS cuenta con una línea de comando, `rostopic list`, que ayuda a saber que *topics* están disponibles y están corriendo en el momento. Una vez iniciado el *rosserial\_python* se debe encontrar en lista los *topics*, índice, corazón, anular, menique y pulgar, como se muestra en la Figura 4.2. Si se

quisiera saber que datos o qué tipo de información tiene dicho *topic* se debe escribir el comando,

```
rostopic echo nombre_del_topic
```

A terminal window titled 'jose@ubuntu: ~' with a dark background. The terminal shows the following text: 'bash: /opt/ros/diamondback/setup.bash: No existe el archivo o el directorio' (twice), 'jose@ubuntu:~\$ rostopic list', and a list of topics: '/anular', '/corazon', '/indice', '/menique', '/pulgarc', '/rosout', '/rosout\_agg'. The prompt 'jose@ubuntu:~\$' is visible at the bottom with a white cursor.

```
jose@ubuntu: ~
bash: /opt/ros/diamondback/setup.bash: No existe el archivo o el directorio
bash: /opt/ros/diamondback/setup.bash: No existe el archivo o el directorio
jose@ubuntu:~$ rostopic list
/anular
/corazon
/indice
/menique
/pulgarc
/rosout
/rosout_agg
jose@ubuntu:~$
```

Figura 4.2. Rostopic list una vez iniciado roserial.  
Fuente: Propia.

Como ejemplo en este caso cada uno de los *topics* recién creados, índice, corazón, anular, menique y pulgar están obteniendo datos del sensor asignado a cada dedo. Por ejemplo para el dedo índice, basta con introducir la siguiente línea de comando y se verá la información como en la Figura 4.3

```
rostopic echo índice.
```



```
data: 489.0
---
data: 495.0
---
data: 493.0
---
data: 487.0
---
data: 480.0
---
data: 476.0
---
data: 476.0
---
data: 483.0
---
data: 489.0
---
data: 494.0
---
data: 493.0
---
data: 487.0
---
data: 478.0
---
data: 476.0
---
data: 478.0
---
data: 484.0
---
data: 492.0
---
data: 495.0
---
data: 492.0
---
```

Figura 4.3. rostopic echo dedo índice.  
Fuente: Propia.

## 4.2.2 Publicadores y suscriptores para Shadow Hand

Para poder realizar la correcta publicación de los ángulos recibidos de los *flex sensor* por medio de Arduino se creó un nodo por cada uno de los dedos, dichos nodos se encuentran suscritos al publicador de Arduino, y a su vez se publican dichos valores en los *topics* de *Shadow Hand* en Gazebo, usados y expresados en la sección anterior.

De esta manera y en forma de ejemplo se muestra el nodo suscriptor y publicador del dedo índice:

Primero que todo se inicializan las variables y se imprime en pantalla los valores que está entregando el *flex sensor* del dedo índice en radianes.

```
#include "ros/ros.h"
#include "std_msgs/Float64.h"
#include <sstream>

std_msgs::Float64::ConstPtr mensaje;
void indiceCallback(const std_msgs::Float64::ConstPtr& msg)
{
    ROS_INFO("Valor indice: [%f]", msg->data);
    mensaje=msg;
}
```

```

int main(int argc, char **argv)
{
    ros::init(argc, argv, "indice");
    ros::NodeHandle n;

```

En esta sección el suscriptor del nodo se suscribe al *topic* "indice" en donde se está publicando la información dentro del nodo *rosserial*.

```

    ros::Subscriber sub = n.subscribe("indice", 50, indiceCallback);

```

En esta sección se publica el valor del ángulo que entrega Arduino a los *topics* de las articulaciones ffj3 y ffj0. Se envía el mismo valor de ángulo a las dos articulaciones debido a que VirtualTouch posee únicamente un sensor por dedo.

```

    ros::Publisher ffj3 =
n.advertise<std_msgs::Float64>("/sh_ffj3_mixed_position_velocity_c
ontroller/command", 50);
    ros::Publisher ffj0 =
n.advertise<std_msgs::Float64>("/sh_ffj0_mixed_position_velocity_c
ontroller/command", 50);
    ros::Rate loop_rate(10);
    ros::spinOnce();

```

Finalmente se publica el valor de la variable mensaje, la cual es el valor del sensor en radianes en ffj3 y ffj0, los cuales son los nombres de los publicadores anteriormente citados.

```

    int count = 0;
    while (ros::ok())
    {

if(mensaje){
    ffj3.publish(mensaje);
    ffj0.publish(mensaje);
}
        ros::spinOnce();
        loop_rate.sleep();
        ++count;
    }

    return 0;
}

```

De este modo se realiza la publicación y suscripción de los nodos para poder entregar dichos valores a Shadow Hand en Gazebo, como se muestra en la

Figura 4.4. Se pueden ver los nodos creados y los *topics* en los que se está publicando o suscribiendo la información, esta imagen se puede obtener con el comando *rxgraph*, que permite ver los nodos que están corriendo en dicho momento y sus respectivas interacciones con otros nodos.

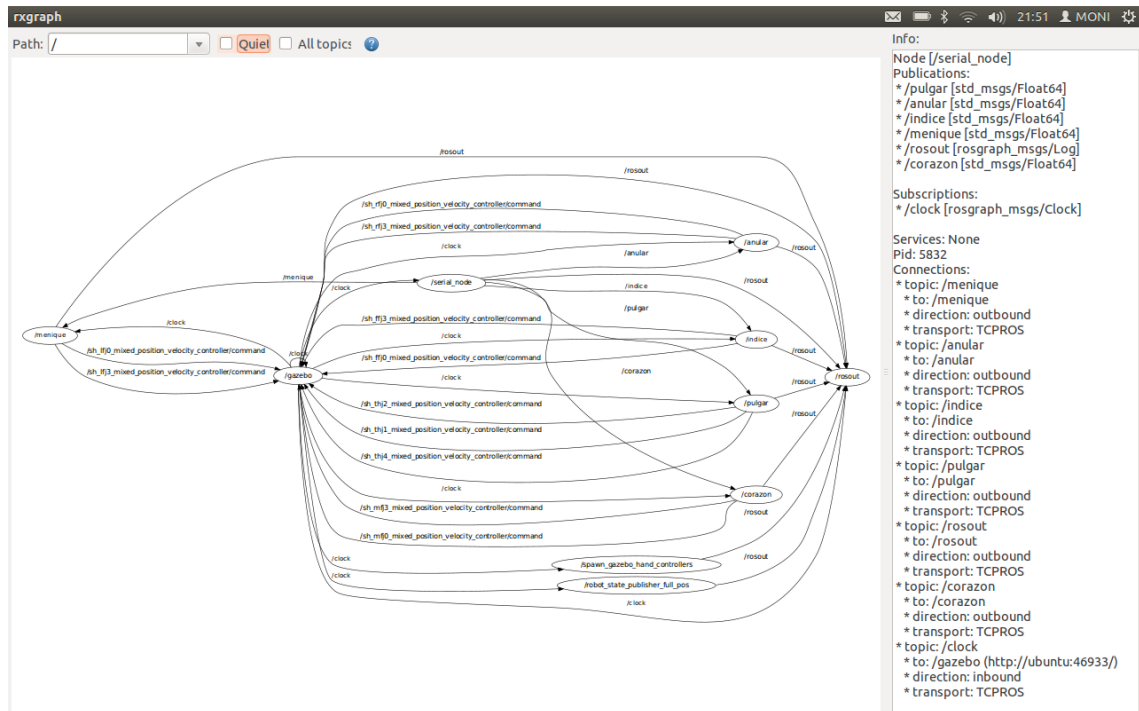


Figura 4.4. Gráfica de comunicación entre Shadow Hand y Arduino.  
Fuente: Propia.

### 4.2.3 Sistema de detección de colisiones

El entorno gráfico de ROS, Gazebo, posee un sistema de simulación de fuerzas naturales para el entorno de desarrollo 3D. Fuerzas como la gravedad, rozamientos, fuerzas externas son simuladas en dicho entorno, Gazebo tiene la opción de crear sensores táctiles en los modelos 3D que se desarrollan en su interior, dichos sensores entregan información como coordenadas de contacto, fuerza de choque, torque y tiempos de duración del mismo. Gracias a esta información se facilitó de forma amplia el desarrollo de la aplicación acortando de este modo la escritura de algoritmos de colisiones como los convencionales realizados en otro tipo de plataformas de desarrollo de software.

El sistema de colisiones de VirtualTouch se desarrolló llevando a cabo los siguientes pasos:

El modelo de *Shadow Robot* posee sensores de contacto, 16 sensores activos, los cuales están ubicados uno en cada falange por dedo y un sensor en la palma de la misma. Para obtener la información de cada uno de los sensores se debe llamar el *topic* que tiene asociado cada sensor. En el caso de la *Shadow Hand* los *topics* de los sensores son los siguientes:

```
/ffdistal_bumper
/ffknuckle_bumper
/ffmiddle_bumper
/rfdistal_bumper
/rfknuckle_bumper
/rfmiddle_bumper
/thdistal_bumper
/thknuckle_bumper
/thmiddle_bumper
/mfdistal_bumper
/mfknuckle_bumper
/mfmiddle_bumper
/lfdistal_bumper
/lfknuckle_bumper
/lfmiddle_bumper
```

De este modo si se hace la llamada al *topic* `ffdistal_bumper`, este proporcionará la información del estado del sensor ubicado en la falange distal del dedo indice, por ejemplo, si se ejecuta el comando:

```
rostopic echo /ffdistal_bumper
```

Y este sensor está siendo colisionado por algún actor externo, el comando anterior brinda información de este tipo:

```
header:
seq: 32213
stamp:
secs: 323
  nsecs: 183000000
  frame_id: thdistal
states:
-
info: touched!
i:0
my geom:thdistal_geom  other geom:ffdistal_geom          time:0

geom1_name: thdistal_geom
geom2_name: ffdistal_geom

wrenches:
-
```

```

force:
  x: 4.3224521471
  y: 1.03446840909
  z: -4.10399759546
torque:
  x: 0.020471011108
  y: -0.0704609036724
  z: 0.00380004778994
-
force:
  x: 0.218667846282
  y: 0.0309026136255
  z: -0.0121754547746

```

De este modo se obtiene la información correcta del estado del sensor, las geometrías que están colisionando, las fuerzas y torques en el punto de colisión. Como se puede ver en el caso anterior se encuentra en la parte inicial en el ítem *info* la palabra “*touched!*” lo cual nos indica que el sensor de dicha falange está siendo colisionado por un actor externo. Más adelante en la sección *force* encontramos las coordenadas en *x*, *y* y *z* de la fuerza aplicada, igualmente en la sección *torque* las coordenadas del torque aplicado. Como se puede ver esta información es útil solo en el momento en que se necesita saber las coordenadas o la fuerza del contacto, por lo cual no es usada en el desarrollo del VirtualTouch.

Si lo que se desea es únicamente obtener un valor de colisión como una bandera, se debe hacer la instalación y llamado del paquete *sr\_tactile\_sensors*, para esto se debe ejecutar el comando de instalación del mismo:

```
GAZEBO=1 rosmake sr_tactile_sensors
```

Y posteriormente una vez lanzado el modelo en Gazebo, se debe correr la línea de ejecución del paquete:

```
roslaunch sr_tactile_sensors sr_tactile_gazebo.launch
```

Este paquete va a proporcionar únicamente la información de la fuerza del contacto, de este modo para el desarrollo de VirtualTouch se hizo uso de este paquete, el cual publica dicha información en los *topics* llamados:

- /sr\_tactile/touch/ff
- /sr\_tactile/touch/mf
- /sr\_tactile/touch/rf
- /sr\_tactile/touch/lf
- /sr\_tactile/touch/th

Una vez conocida la información del contacto de cada uno de los sensores, que en el caso de VirtualTouch se utilizan únicamente los sensores ubicados en la yema de los dedos, ya que es donde físicamente se encuentran ubicados los actuadores, se desarrolló un modelo suscriptor-publicador simple, el cual se suscribe al *topic* del paquete *sr\_tactile\_sensor* para obtener la información del contacto y posteriormente se publica dicha información en un *topic* creado desde Arduino, en donde se establece como 0 para no colisión y un valor diferente de cero como colisión. De este modo se activa o se desactiva la salida de Arduino asociada a cada uno de los vibradores que posee el guante.

#### 4.2.4 Adecuación del entorno virtual

Inicialmente al abrir y cargar Gazebo y *Shadow Hand*, visualmente no presentan una ubicación adecuada para la utilización de esta, debido a que la cámara tiene un valor por defecto en Gazebo, y aunque el entorno permite un libre movimiento dentro de él, no es cómodo tener que acomodarla cada vez que se cargue el programa. Es por esto que se realizaron modificaciones para lograr una buena ubicación de la cámara y una posición cómoda de *Shadow Hand* desde un principio.

Para esto se realizó una copia del archivo base del mundo de Gazebo `empty.world` y `empty_world.launch` que se encuentran en `opt/ros/electric/simulator_gazebo/loaders/worlds` y se copió en el espacio de trabajo creado para la instalación de *Shadow Robot virtualtouch/launch/gazebo\_loaders*. En realidad se puede poner en la ubicación que se desee, la copia de los archivos debe hacerse debido a que los archivos originales se encuentran en carpetas del sistema y este no permite hacer modificaciones. Dentro de `empty.world` se encuentra toda la descripción de un entorno vacío dentro de Gazebo, están características como la gravedad, el tamaño de la pantalla, especificaciones de la renderización, ubicación y zoom de la cámara entre otras; específicamente en este archivo se hicieron cambios en la ubicación y zoom de la cámara.

Una vez logrado la ubicación y zoom de la cámara deseada, se prosigue con buscar una posición de *Shadow Hand* adecuada para el uso de VirtualTouch, recordando que no se tiene implementado los movimientos de la muñeca. Dentro del archivo que se utiliza para cargar la *Shadow Hand* en Gazebo (`gazebo_arm_and_hand.launch`), es donde se especifican las posiciones de cada una de las partes de esta, aquí inician cargando el soporte, luego el brazo y finalmente el antebrazo y la mano que vienen en una sola pieza, creando así dos juntas, *ShoulderJswing* y *ElbowJswing*, que hacen

referencia a las juntas del hombro y codo respectivamente. Contiguo a cada una hay un valor, este valor corresponde a un ángulo en radianes, para este ángulo debe tomarse en cuenta el eslabón anterior, es decir que para cada eslabón se crea una nueva referencia del plano cartesiano xyz. Los valores iniciales que este tiene son 0.78 y 2.0 respectivamente, que se cambian a 0.35 y 1.2 respectivamente.

Una vez ubicada la mano como se desea, es necesario crear el entorno. Este entorno deberá contar con objetos que permitan la detección de colisiones, es por esto que se aumenta al entorno una mesa, a una altura adecuada, y sobre ella y frente a la mano un objeto. Finalmente el entorno se verá como en la Figura 4.5.

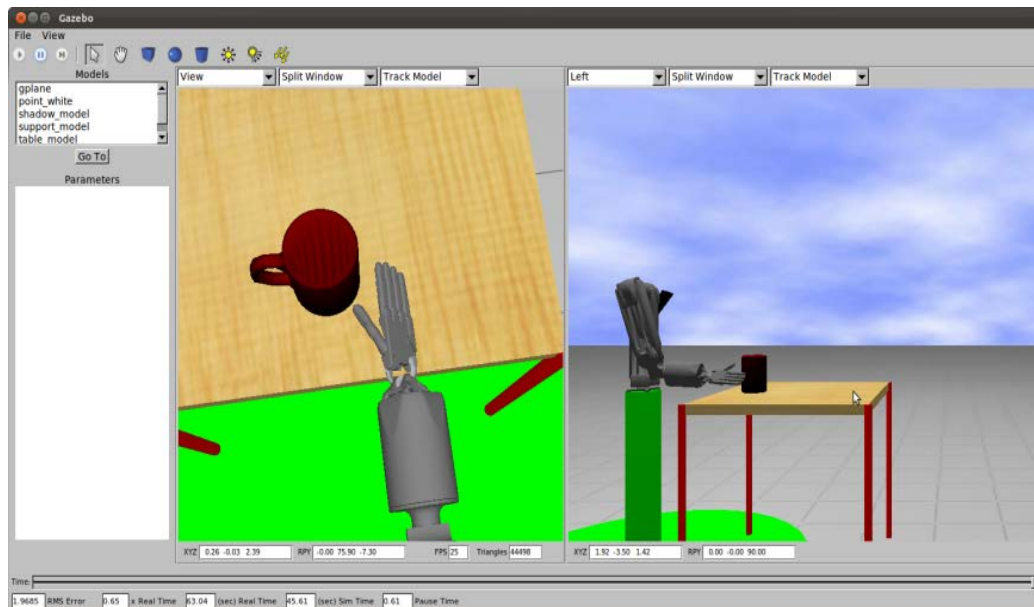


Figura 4.5. Entorno virtual para VirtualTouch.  
Fuente: Propia.

Finalmente es necesario tener un archivo “.launch”, llamado en este caso `virtualtouch.launch`, en donde se llaman los nodos y archivos necesarios, para la utilización de VirtualTouch. Se llamara a la *Shadow Hand* modificada anteriormente, el entorno de Gazebo, y finalmente la mesa y la taza a utilizar. En este archivo también se llamaran a los nodos creados con anterioridad para el funcionamiento con Arduino. En la Figura 4.6 se puede observar el guante siendo usado. El programa se correrá, escribiendo la siguiente línea de comando en una terminal nueva:

```
roslaunch virtualtouch virtualtouch.launch
```

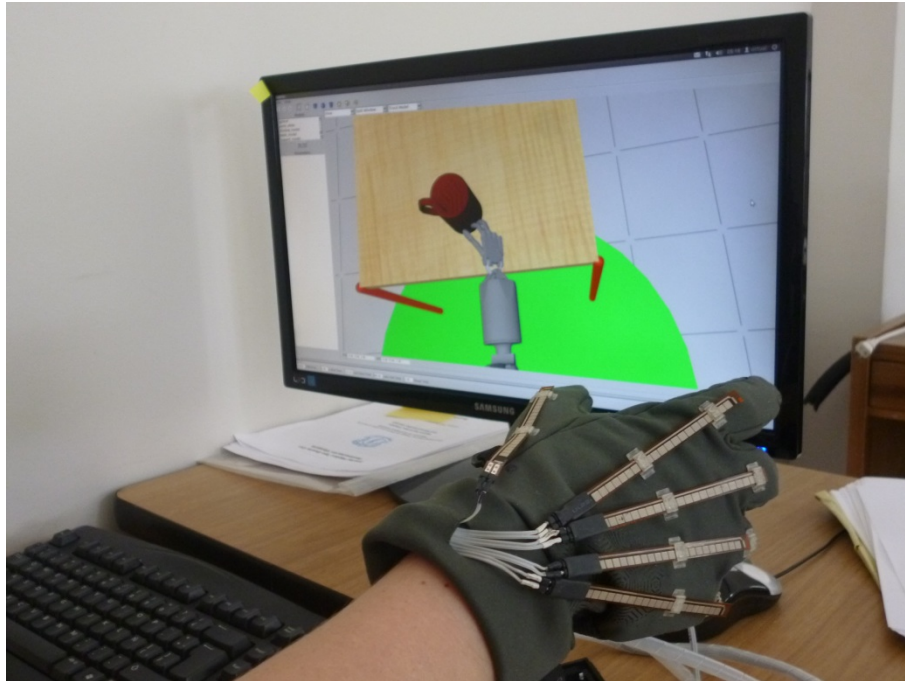


Figura 4.6. Prueba usando VirtualTouch en ROS.  
Fuente: Propia.

### 4.3 DESARROLLO DEL SOFTWARE EN QT/VTK

Como ya se mencionó anteriormente, ROS y Gazebo no contaban con la implementación de objetos deformables, por lo que se hizo uso de VTK, QT y VCollide para cumplir con los objetivos del proyecto. De manera similar se tiene entonces, el desarrollo de una interfaz para el usuario, la adquisición de datos (comunicación interfaz-Arduino), detección de colisiones y por último la deformación de objetos.

#### 4.3.1 Desarrollo de la interfaz de usuario

Se inicia entonces con el desarrollo de la interfaz de usuario en *QT Creator*, esta interfaz es una ventana muy sencilla, se tiene un cuadro para el renderizado de los objetos 3D, la mano y el objeto deformable. Estos objetos están modelados en *Blender* y exportados en el formato *wavefront (.obj)*, que es una extensión que permite guardar la geometría 3D del objeto, como la posición de vértices, texturas, coordenadas y polígonos. La interfaz cuenta con un botón que permite iniciar la comunicación con Arduino (Figura 4.7).



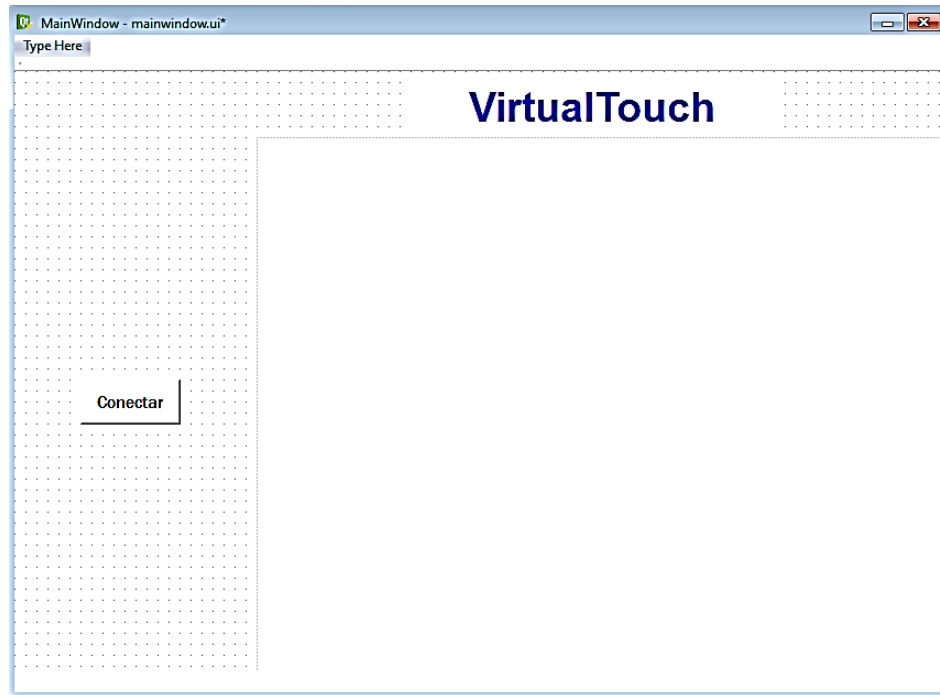


Figura 4.7. Interfaz creada en QT Creator.  
Fuente: Propia.

Una vez creada la interfaz de usuario se procede a la programación e integración con VTK para la renderización de los modelos 3D. VTK hace uso de *pipelines* (Figura 4.8) como estrategia de solución, demandando datos en forma secuencial hasta llegar a la salida deseada por el usuario.



Figura 4.8. Pipeline de VTK para renderizado.  
Fuente: Propia.

Por consiguiente para la visualización y tratamiento visual de la mano se creó la clase *Mano* la cual cuenta con toda la descripción y especificaciones de la misma teniendo en cuenta el *pipeline* de la Figura 4.8, como ejemplo se ilustra como cargar un objeto *wavefront(.obj)*, para posteriormente ser renderizado.

Se inicia cargando el objeto *wavefront(.obj)* que se desea renderizar, este vendría siendo la fuente de la *pipeline*. Para leerlo es necesaria la librería *vtkOBJReader*. Como filtro se tiene la librería *vtkPolyData*, esta librería representa un conjunto de datos como líneas, polígonos y triángulos, al que se le pasa el archivo *.obj* creado anteriormente. Se continúa con

vtkPolyDataMapper, que genera primitivas gráficas a partir de datos poligonales, y a este se le asigna el polyData recién creado. Posteriormente se asigna el Mapper al vtkActor.

```
file="D:/Deformacion/menique.obj";
obj = vtkOBJReader::New();
polyData = vtkPolyData::New();
mapper = vtkPolyDataMapper::New();
actor = vtkActor::New();
obj->SetFileName(file[j]);
obj->Update();
polyData = obj->GetOutput();
mapper->SetInput(polyData);
actor->SetMapper(mapper);
```

Este proceso se hace para cada uno de los obj's. En este caso para cada una de las falanges y la palma, por lo que antes de renderizar se debe realizar el ensamblado de las falanges con la ayuda de vtkAssembly, las uniones se hicieron teniendo en cuenta las articulaciones verdaderas. Por ejemplo en el código siguiente se crea un articulación para índice1 e índice 2 que hacen referencia a las falanges proximal y distal del dedo índice:

```
articulaciones[1] = vtkAssembly::New();
articulaciones[1]->AddPart(actor[4]); //indice1
articulaciones[1]->AddPart(actor[3]); //indice2
```

Debido a que posteriormente se les dará movimiento a cada falange es necesario especificar sobre que eje de coordenadas (xyz) va a realizar la rotación, por lo que debe asignársele a cada actor (falange) y las articulaciones creadas su origen, es decir el punto de rotación como se muestra a continuación. Los valores son obtenidos mediante las coordenadas (coordenada máxima y mínima) de la ubicación de cada una de las falanges en el espacio x, y, y z:

```
actor[1]->SetOrigin(4.272,2.824,3.001); //Pulgar
actor[2]->SetOrigin(4.619,4.154,2.321); //Indice
actor[3]->SetOrigin(4.876,4.313,1.598); //Corazon
actor[4]->SetOrigin(4.987,4.157,1.066); //Anular
actor[5]->SetOrigin(5.103,3.849,1.108); //Meñique
```

Finalmente se renderiza la mano y de manera similar el objeto deformable con `renderer->AddActor(actor1)`, obteniendo entonces la interfaz final como se muestra en la Figura 4.9.

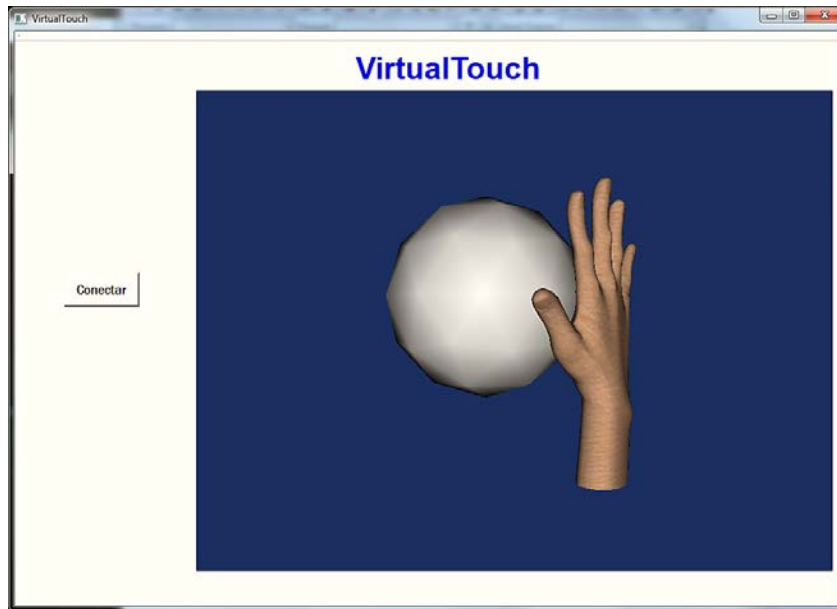


Figura 4.9. Interfaz VirtualTouch en VTK/QT.  
Fuente: Propia.

### 4.3.2 Comunicación de la interfaz con Arduino

La comunicación realizada para el Guante VirtualTouch se implementó buscando la mayor semejanza posible entre el movimiento real y el movimiento del modelo 3D. Se hizo necesario de igual manera implementar en la programación de la tarjeta las ecuaciones que permitieran entregar los valores en grados de  $0^\circ$  a  $90^\circ$ . De igual manera se implementó una alerta para la realimentación al guante, es decir que en el momento de colisión, el vibrador fuese accionado.

Para la comunicación se utilizó la clase *TSerial* que es la que permite leer y/o enviar información del puerto serial activado y configurado para dicho fin. De esta forma se pueden recibir y enviar datos de y hacia Arduino. La tarjeta Arduino hay que programarla de igual forma para que escriba y reciba información del puerto serial usando las funciones *Serial.write* y *Serial.read* de Arduino. Se debe tener en cuenta que Arduino envía los valores organizados en un vector por lo que debe recibirse como un vector, por consiguiente hay que tener cuidado en asignar el valor correcto a cada falange. A continuación se muestra la configuración del puerto serial y la lectura de los datos, la definición del vector que recibe los datos debe ser de igual tamaño al que se está enviando desde Arduino, sino se generan errores, así como se muestra el vector `rx_datos[5]` y la lectura `getArray`, en donde guarda los datos de Arduino uno por uno en el vector de 5 espacios:

```

com->disconnect();
com->connect("COM3",9600,spNONE);
char rx_datos[5];

com->getArray(rx_datos,5);

```

Los datos adquiridos son enviados a cada una de las falanges, teniendo en cuenta el orden en que se reciben y el eje de rotación (x, y y z). Para las falanges índice, corazón, anular y meñique es el eje z, mientras que para el pulgar es el eje y y debido a la naturaleza del movimiento del mismo. El orden de estas coordenadas puede variar dependiendo de las coordenadas con las que se haya creado, guardado y exportado en Blender. La rotación en sí se hace sobre el eje que atraviesa transversalmente cada una de las falanges, como se puede mostrar en la Figura 4.10 y reflejado en el código que se muestra a continuación, donde los valores recibidos desde Arduino son enviados al eje y para el pulgar y el eje z para las demás falanges:

```

hand->actor[1]->SetOrientation(0,-rx_datos[0],0); //Pulgar
hand->actor[2]->SetOrientation(0,0,rx_datos[1]); //índice
hand->actor[3]->SetOrientation(0,0,rx_datos[2]); //corazon
hand->actor[4]->SetOrientation(0,0,rx_datos[3]); //anular
hand->actor[5]->SetOrientation(0,0,rx_datos[4]); //menique

```

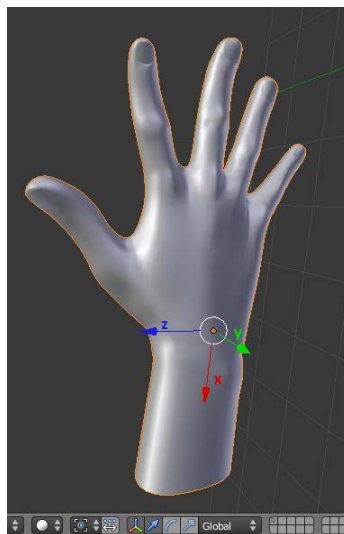


Figura 4.10. Eje de coordenadas de la mano 3D en Blender.  
Fuente: Propia.

### 4.3.3 Detección de colisiones

Una vez lograda la comunicación entre Arduino con VTK y QT se inicia la integración de VCollide con VTK. Se tomaron como trabajos base [23] y [24] en los que se hace la integración de estas dos librerías activando o desactivando las colisiones entre pares de objetos y generando el reporte de las mismas. Se toma la clase “*DeteccionColisiones*” de [24] (Figura 4.11)

donde se muestran los elementos necesarios para esta clase, el objeto hace referencia a todos los objetos que van a colisionar, el elemento matriz permite obtener la posición y orientación del objeto que va a colisionar (en este caso cada una de las falanges ensambladas), y el dato booleano permite saber si existe colisión entre los objetos.



Figura 4.11. Clase DeteccionColisiones.  
Fuente: [24].

Con el Diagrama de flujo de la Figura 4.12, se puede entender mejor aún el orden del funcionamiento de la clase.

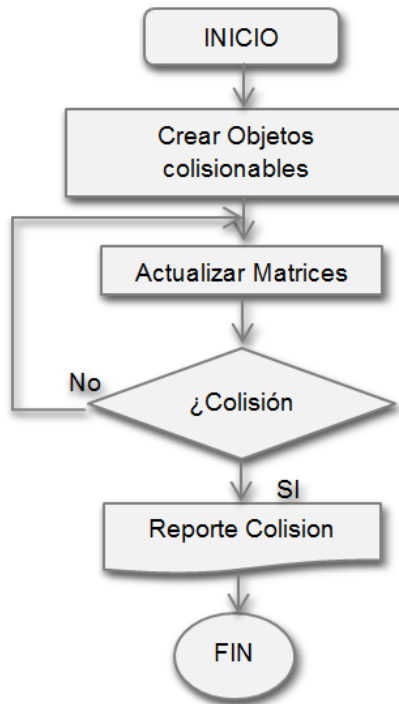


Figura 4.12. Diagrama de Flujo de detección de colisiones.  
Fuente: Propia.

La creación de objetos colisionables se muestra en código a continuación, los polyData referenciados son las falanges distales del pulgar, índice, corazón, anular y meñique respectivamente, el método `SetObjeto` permite crear la triangularización de los polígonos de cada objeto:

```
colisiones->SetObjeto(Limpieza->GetOutput());
```

```
colisiones->SetObjeto(hand->polyData[1]);  
colisiones->SetObjeto(hand->polyData[4]);  
colisiones->SetObjeto(hand->polyData[7]);  
colisiones->SetObjeto(hand->polyData[10]);  
colisiones->SetObjeto(hand->polyData[13]);
```

La actualización de matrices se hace con el método `ActualizarMatriz` (`int VCollide_ID, vtkMatrix4x4 *matriz`), este nos permite actualizar la matriz (información de posición y rotación) de cada objeto. El primer parámetro hace referencia a un ID creado para cada objeto colisionable, para diferenciar los actores, y el segundo es la matriz de dicho actor.

La colisión y el reporte de colisión estará dado por el método `GetIdCeldaContacto()`, el cual nos permite saber en qué parte del objeto se produjo la colisión, brinda información sobre las celdas (triángulos) que han realizado contacto

En las Figuras 4.13 y 4.14 se puede ver cuando hay o no colisiones.

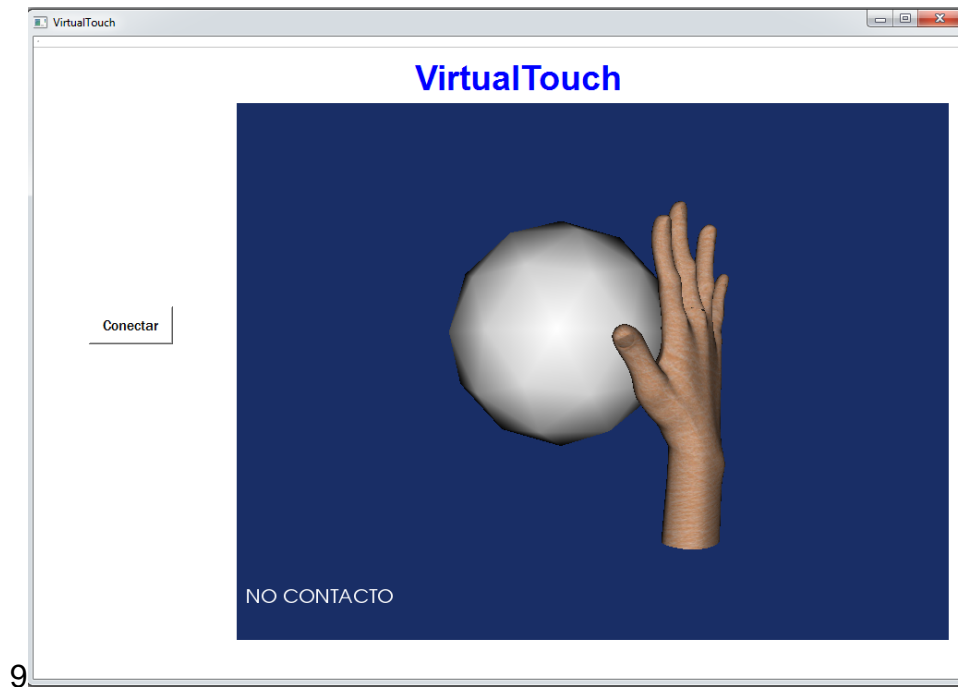


Figura 4.13. Visualización no contacto.  
Fuente: Propia.

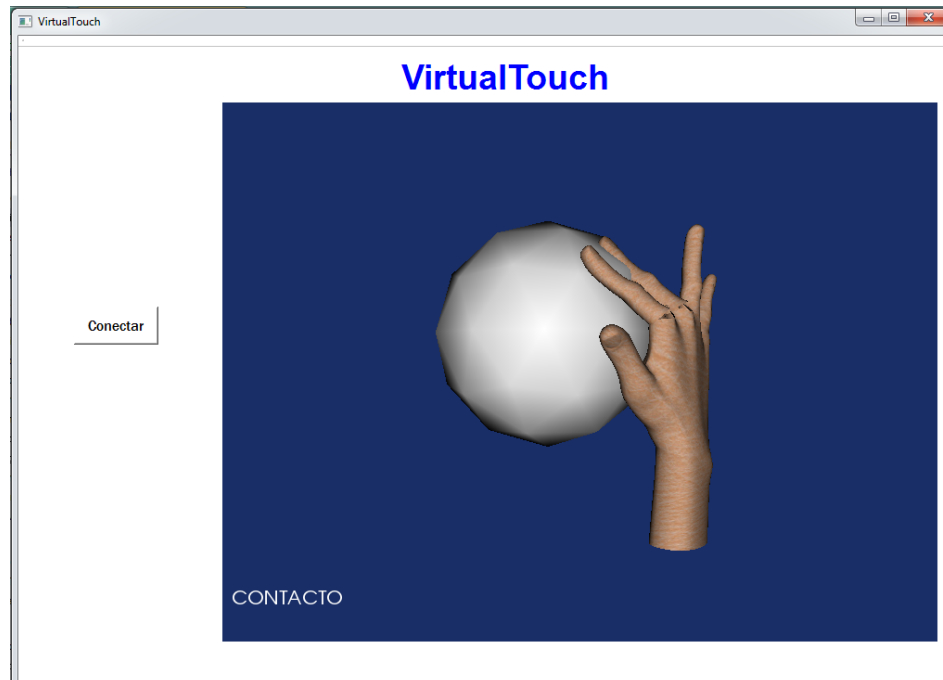


Figura 4.14. Visualización de contacto.  
Fuente: Propia.

#### 4.3.4 Deformación de objetos

Por último se implementó la deformación de objetos basado de igual manera en el trabajo realizado por [24], en el cual se crearon las clases necesarias para la deformación usando un sistema de resortes. En la Figura 4.15 se muestra la estructura planteada.

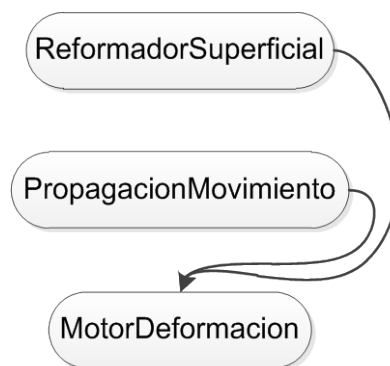


Figura 4.15. Estructura del motor de deformación.  
Fuente: [24].

Esta estructura tiene tres clases para el objetivo de la deformación conformado por ReformadorSuperficial, PropagacionMovimiento, MotorDeformacion. ReformadorSuperficial contiene los datos de las

coordenadas de los vértices del objeto 3D, por tanto la malla del objeto está formada por esta información. Estos datos podrán ser modificados por la clase `MotorDeformacion`. `PropagacionMovimiento` entrega información de los vértices que serán afectados por la deformación en los diferentes niveles de propagación. `MotorDeformacion` define el cómo se deben modificar las coordenadas de los vértices de acuerdo al modelo deformable [24].

Se presenta el código que representa el esquema anterior aplicado al proyecto. La primera parte es de la clase `ReformadorSuperficial`, en la variable `reformador`, esta clase modifica los vértices del objeto3D de tal forma que se puede moldear su superficie, estos vértices son almacenados en un vector que se modifica cada vez que es necesario, esta clase también elimina los bordes afilados del objeto 3D [24].

```
reformador->SetEntrada( Limpieza->GetOutput() );  
reformador->GetSalida();  
mapper->SetInput( reformador->GetSalida() );
```

La segunda parte viene dada por la clase `PropagacionMovimiento` definido en la variable `propagacion` la cual toma un objeto y organiza sus vértices de tal forma que exista una propagación de vértices permitiendo la deformación, la deformación implementada está basado en un sistema de resortes como se explica en [24], donde el algoritmo recorre la malla desde el vértice afectado, afectando consecutivamente los demás vértices.

```
propagacion->SetEntrada( Limpieza->GetOutput() );  
propagacion->GetPropagacionVertices();
```

Esta tercera y última parte es de la clase `MotorDeformacion`, la cual tiene dos métodos básicos, el método de `Deformar` que se encarga de que el objeto se deforme y el método `Restaurar` que recupera la forma original del objeto. [24]

```
deformacion->SetEntrada( Limpieza->GetOutput() );  
deformacion->SetReformador(reformador);  
deformacion->SetPropagacionVerts(propagacion->  
GetPropagacionVertices());  
deformacion->Deformar();  
deformacion->RestaurarForma();
```

En la Figura 4.16 se puede observar la prueba de la deformación, en este caso realizada por el contacto del dedo pulgar, finalmente en la Figura 4.17 se muestra el guante `VirtualTouch` siendo usado en la plataforma creada con QT y VTK.



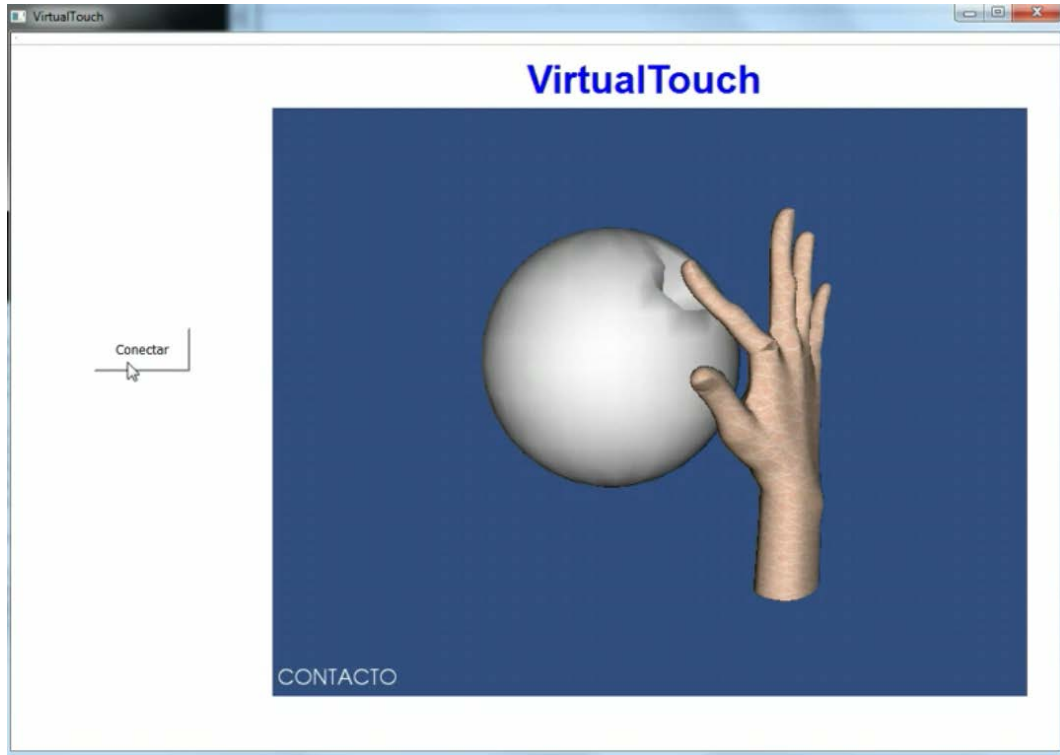


Figura 4.16. Prueba VirtualTouch con Deformación.  
Fuente: Propia.

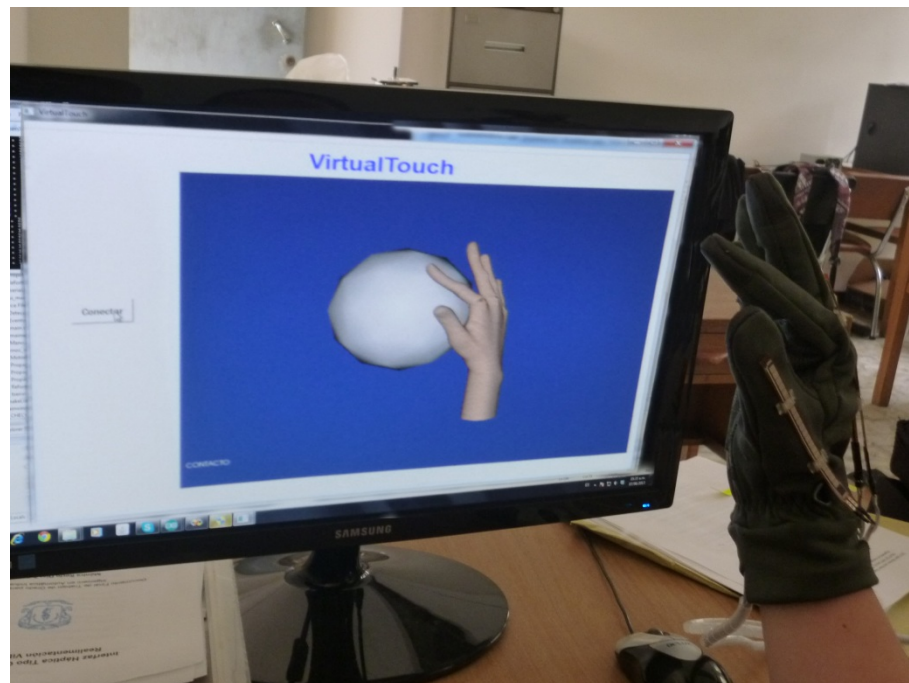


Figura 4.17. Utilización de VirtualTouch con QT/VTK.  
Fuente: Propia.

## 5. CONCLUSIONES Y TRABAJOS FUTUROS

### 5.1 CONCLUSIONES

VirtualTouch es el primer guante háptico con realimentación vibratoria que cuenta con el sensado de todos los dedos bajo herramientas de software libre, desarrollado en el país y por ende en la Universidad del Cauca. Es además el segundo proyecto desarrollado bajo el convenio OpenSurg en la Universidad Miguel Hernández de Elche, España.

Al finalizar este trabajo se pudo concluir que ROS es una plataforma muy versátil y útil en trabajos de investigación y académicos, ya que por su filosofía se reducen tiempos en implementación y permite abordar problemas de un alto nivel, además de permitir el desarrollo software bajo lenguajes de programación como *Python* y *C++* que son los lenguajes más utilizados y fáciles de utilizar, aunque ROS permite la integración de otras librerías externas. Otra ventaja de ROS es que fomenta el trabajo en equipo y establece convenios, procesos y metodologías para hacer software reutilizable con otros grupos de investigación. ROS presenta como desventajas la gran variedad de *stacks* o paquetes, según el autor, o inclusive la versión de ROS y de Ubuntu, debido a la continua y rápida evolución que ROS presenta; y por otra parte la curva de aprendizaje es lenta, por su misma filosofía y el correcto manejo de sus conceptos.

Las librerías QT, VCollide y VTK son de igual manera muy versátiles, demostrado en la facilidad con que se integraron las tres para lograr un objetivo. Aunque VCollide no sea la única librería existente para la detección de colisiones es quizás la más sencilla de utilizar, sin embargo genera una carga computacional alta y disminuye el rendimiento del programa.

### 5.2 TRABAJOS FUTUROS

- Se propone lograr la deformación de objetos con VirtualTouch una vez ROS y Gazebo, cuenten con la herramienta.
- Existen otras librerías y herramientas, para la detección de colisiones y deformación como Physx, que a pesar de sus limitaciones para la programación ya que se debe contar con una tarjeta gráfica Nvidia para su desarrollo, presenta buenos resultados computacionalmente. Sería conveniente realizar un software para la utilización del guante con este tipo de herramientas.

- Implementar la mano izquierda tanto para la parte hardware como software, debido a las diferentes necesidades de los posibles usuarios y una mayor simulación de la realidad.
- Es necesario la búsqueda de otro tipo de realimentación para el guante que se asemeje un poco más a la realidad, ya que el incorporado en esta versión, da una alerta más que una sensación real.
- Implementar más sensores a VirtualTouch, uno por cada falange en cada dedo y sensores en las aberturas de los dedos para así simular de manera más real los movimientos de la mano, así como en la muñeca para simular los movimientos de la misma.
- Implementar un software más avanzado con el fin de enfocar el uso de VirtualTouch a actividades orientadas a la medicina.

## 6. REFERENCIAS

- [1] Youngblut Christine, Jhonson Rob, Sara Nash, Ruth Wienclaw, and Will Craig, *Review of virtual environment interface technology.*: IDA Paper, Marzo 1996, ch. 6, pp. 125-144.
- [2] Margaret Minsky, Ming Ouh-young, Oliver Steele, Frederick Brooks, and Max Behensky, "Feeling and Seeing: Issues in Force Display," *ACM Siggraph Computer Graphics*, vol. 24, no. 2, pp. 235-241, 1990.
- [3] Manuel Ferre, Maria Oyarzabal, Alexandre Campos, and Mary Monroy, "Multifinger Haptic Interfaces for Collaborative Environments," *Haptics: perception, devices and scenarios. Proceedings of 6th International Conference, EuroHaptics 2008*, pp. 101-112, 2008.
- [4] Geomagic. (2012, Marzo) Geomagic. [Online]. <http://geomagic.com/es/>
- [5] Sensable Technologies. (2009, Julio) Specifications for the PHANTOM Desktop and PHANTOM Omni Haptic devices. Especificaciones del Producto.
- [6] Sensable Technologies. (2006, Enero) Specifications for the PHANTOM premium 3.0/6DOF haptic Device. Especificaciones del producto.
- [7] Nicholas Patrick, *Design, Construction and Testing of a Fingertip Tactile Display for interaction with Virtual and Remote Environments.*: Master thesis of Science in Mechanical Engineering, Department of Mechanical Engineering, MIT, 1990.
- [8] W.S. Harwin. (2012, Marzo) Introduction to haptics and haptic technologies. [Online]. <http://www.personal.rdg.ac.uk/~shshawin/LN/L8hapticdesigns.html>
- [9] CyberGlove Systems. (2012, Marzo) Cyberglovesystem. [Online]. <http://www.cyberglovesystems.com/>
- [10] 5DT (Fifth Dimension Technologies). (2012, Marzo) 5DT Fifth Dimension Technologies, Virtual reality for the real World. [Online]. <http://www.5dt.com/products/pdataglove5u.html>
- [11] Direct Industry. (2012, Octubre) Direct Industry, El salon virtual de la industria. [Online]. [www.directindustry.es](http://www.directindustry.es)

- [12] Sensor Workshop at ITP. (2012, Octubre) [Online]. <http://itp.nyu.edu/physcomp/sensors/Reports/Flex>
- [13] Precision Microdrivers. (2012, Octubre) [Online]. <https://catalog.precisionmicrodrives.com/order-parts/product/310-113-10mm-vibration-motor-3-4mm-type>
- [14] ARDUINO. (2012, Octubre) ARDUINO. [Online]. <http://arduino.cc/>
- [15] CadSoft Computer. (2012, Octubre) CadSoft. [Online]. <http://www.cadsoftusa.com/>
- [16] Saul Gausin. (2013, Mayo) Computo Integrado. [Online]. <http://computointegrado.blogspot.com/2012/04/uso-de-flex-sensor-con-arduino.html>
- [17] Autodesk Inc. (2012, Octubre) AUTODESK. [Online]. <http://www.autodesk.com/products/autodesk-inventor-family/overview>
- [18] Alava ingenieros. (2012, Octubre) Tecnología a su medida. [Online]. <http://www.alava-ing.es/ingenieros/productos/tecnologias-de-prototipado-3d-y-simulacion/impresoras-3d/impresora-3d-systems-projet-hd-3000/>
- [19] ROS. (2012, Octubre) [Online]. [www.ros.org](http://www.ros.org)
- [20] Yale School of Medicine. (2013, Marzo) [Online]. <http://www.yalemedicalgroup.org/stw/Page.asp?PageID=STW025147>
- [21] Qt Project. (2013, Enero) [Online]. <http://qt-project.org/>
- [22] Kitware. (2013, Enero) VTK - The Visualization Toolkit. [Online]. <http://www.vtk.org/>
- [23] Daniel Ivorra, "Simulador Háptico para entrenamiento de técnica de laparoscopia", Universidad Miguel Hernandez de Elche, Elche, España, Proyecto fin de carrera Ingeniería de Telecomunicación Junio 2008.
- [24] Faber Montero and Raul Gomez, "VISUALIZACIÓN Y DEFORMACIÓN DE OBJETOS VIRTUALES 3D," Universidad Del Cauca, Popayán, Tesis de Ingeniería en Automática Industrial 2012.