



Universidad de Valladolid



ESCUELA DE INGENIERÍAS
INDUSTRIALES

UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERIAS INDUSTRIALES

**Grado en Ingeniería Electrónica Industrial y
Automática**

**Desarrollo de un interfaz de usuario
para guante de datos 5DT Data Glove**

Autor:

San José Burgos, Pablo

Tutor:

**González Sánchez, José Luis
Departamento de Ingeniería de
Sistemas y Automática**

Valladolid, Enero de 2017

Resumen

El objetivo de este TFG es el desarrollo de un entorno de realidad virtual en el que a través de el modelo de una mano se representen los datos de flexión de los dedos de una mano real que esté usando un guante “5DT Data Glove 14 Ultra Left”. Para ello se debe además desarrollar toda la capa de comunicación entre el guante y el entorno de realidad virtual. Esta capa de comunicación debe incluir la publicación de los valores leídos en el programa Robotic Operating System (ROS) con un paradigma publicador/subscriptor. Todo ello se encuentra enmarcado en el proyecto “Sistema robotizado colaborativo para cirugía laparoscópica asistida por la mano” desarrollado por la Universidad de Valladolid, la Universidad Miguel Hernández de Elche y la Universidad de Málaga.

Palabras Clave

- Guante de datos
- Interfaz de usuario
- Interfaz Gráfico
- ROS
- Comunicación

Abstract

The goal of this Project is the development of a virtual reality environment in which, through the model of a hand, flexure data read from a sensorized “5DT Data Glove 14 Ultra Left” is represented. This includes developing the communication layer between the glove and the VR Environment. This Layer must include a publisher/subscriber paradigm implemented with “Robotic Operating System” or ROS. This Project is part of a greater one called “ Sistema robotizado colaborativo para cirugía laparoscópica asistida por la mano” which is in development by the University of Valladolid, the University Miguel Hernández of Elche and the University of Málaga.

Keywords

- Data glove
- User interface
- Graphical interface
- ROS
- Communication

Acrónimos

- ROS: Robotic Operating System
- GUI: Graphical User Interface
- HALS: Hand Assisted Laparoscopic Surgery
- OOP: Object Oriented Programming
- TCP: Transmission Control Protocol
- UDP: User Datagram Protocol
- IP: Internet Protocol
- VR: Virtual Reality
- RT: Real Time
- JSON: JavaScript Object Notation
- GDL: Grado de libertad
- 3D: Three Dimensions
- API: Application program interface
- LUT: Look-Up Table
- GNU: GNU's Not UNIX
- CC: Creative Commons

Índice de contenido

1	Introducción y objetivos.....	15
1.1	Marco del trabajo.....	15
1.1.1	Introducción a la realidad virtual y aumentada.....	15
1.1.2	Estado del arte.....	17
1.1.3	Introducción a la cirugía laparoscópica y HALS.....	19
1.1.4	Marco del proyecto: Sistema robotizado colaborativo para cirugía laparoscópica asistida por la mano.....	22
1.2	Objetivos.....	24
1.2.1	Objetivo global.....	24
1.2.2	Objetivos parciales.....	24
1.3	Planteamiento del TFG.....	25
1.3.1	Materiales utilizados y descartados.....	26
1.3.2	Modificaciones.....	31
2	Desarrollo del TFG.....	35
2.1	Hito 1.....	35
2.2	Hito 2.....	45
2.2.1	Justificación del interfaz ROS-Entorno virtual.....	45
2.2.2	Relación y descripción de los ficheros generados.....	46
2.3	Hito 3.....	57
2.3.1	Estudio preliminar de los requisitos.....	57
2.3.2	Modelo 3D en Blender.....	58
2.3.3	Creación de la armadura.....	61
2.3.4	Programación del interfaz.....	69
2.3.5	Hilos de procesamiento.....	73
2.4	Hito 4.....	81
2.4.1	Motivación.....	81
2.4.2	El interfaz.....	82
2.5	Funcionamiento integrado.....	84
2.5.1	Lanzamiento de la aplicación.....	85
2.5.2	Etapa 1 - Publicación de los datos.....	87
2.5.3	Etapa 2 - Suscripción a los datos y representación gráfica.....	90
3	Resultados.....	93
3.1	Velocidad de reacción.....	93
3.2	Uso de recursos.....	93
3.3	Pérdida de información.....	96
4	Líneas futuras.....	97
5	Conclusiones.....	99
6	Bibliografía.....	101
Anexos	105
Anexo I	107
Anexo II	111
Anexo III	123
Anexo IV	143
Anexo V	153
Anexo VI	155

Anexo VII.....	157
Anexo VIII.....	169
Anexo IX.....	175
Anexo X.....	177
Anexo XI.....	181

Índice de figuras

Figura 1: Virtual-Reality Continuum.....	15
Figura 2: Imagen de una operación HALS. Cortesía del World Laparoscopic Hospital.....	20
Figura 3: Operación en entorno asistido por Robot. Cortesía de Víctor Muñoz..	21
Figura 4: Fotografía del entorno de desarrollo.....	22
Figura 5: Esquema general de la aplicación.....	24
Figura 6: Tabla de especificaciones para guantes 5 Ultra y 14 Ultra.....	26
Figura 7: Estructura de ROS. Cortesía de Open Source Robotics Foundation....	26
Figura 8: Función de ROS en el paradigma Publicador/Subscriptor.....	27
Figura 9: Estructura Topic-Chats.....	28
Figura 10: Shadowhand.....	29
Figura 11: Portada de Blender 2.78a, la versión utilizada.....	31
Figura 12: Fórmula de la resolución en bits.....	35
Figura 13: Muestreo digital de una señal analógica.....	35
Figura 14: Fórmula de la calibración.....	37
Figura 15: Ejemplo de valores de ángulo.....	37
Figura 16: Estructura del paquete del protocolo.....	39
Figura 17: Esquema temporal de la comunicación en el Hito 1.....	40
Figura 18: Proceso de funcionamiento del lanzamiento de Listeners.....	47
Figura 19: Huesos y articulaciones de la mano derecha, vista dorsal.....	57
Figura 20: Partes del hueso en Blender.....	61
Figura 21: Modelo de la mano, con la cámara y el punto de luz fijados.....	62
Figura 22: Proceso de división del hueso de un dedo.....	63
Figura 23: Esquema de parentescos y dependencias.....	64
Figura 24: Detalle relación de parentesco con offset.....	65
Figura 25: Detalle ejes locales de los huesos de dos dedos.....	65
Figura 26: Detalle "Weight paint" en la falange distal del dedo índice.....	66
Figura 27: Tabla GDL de cada hueso.....	67
Figura 28: Detalle editor lógico de Blender.....	69
Figura 29: Proceso realizado por los hilos.....	69
Figura 30: Detalle editor de propiedades del juego de Blender.....	71
Figura 31: Algoritmo de posicionamiento.....	77
Figura 32: Proceso de ejecución de un comando desde el GUI.....	81
Figura 33: Botonera GUI desarrollada con Tkinter.....	83
Figura 34: Árbol de directorios propios del proyecto.....	84
Figura 35: Terminal de compilación.....	85
Figura 36: Proceso de arranque.....	86
Figura 37: Terminal roscore.....	86
Figura 38: Terminal mostrando el proceso de calibración.....	87
Figura 39: Terminal mandando la calibración.....	88
Figura 40: Terminal recibiendo la calibración.....	88
Figura 41: Menú de lanzamiento del interfaz de Blender.....	90
Figura 42: Mano renderizada en pose inicial.....	91
Figura 43: Uso de CPU y memoria no cacheada en reposo.....	93
Figura 44: Desglose de memoria en reposo.....	93
Figura 45: Uso de CPU y memoria no cacheada en funcionamiento.....	94
Figura 46: Desglose de memoria en funcionamiento.....	94

1 Introducción y objetivos

En esta sección se engloba el marco de trabajo del proyecto, tanto a nivel de estado del arte como dentro del proyecto general al que pertenece, y se describen los objetivos particulares de este Trabajo de Fin de Grado

1.1 Marco del trabajo

1.1.1 Introducción a la realidad virtual y aumentada

"Realidad virtual: un sistema de computación usado para crear un mundo artificial en el cual el usuario tiene la impresión de estar y la habilidad de navegar y manipular objetos en él". Manetta C. y R. Blade (1995)

"Realidad aumentada: forma de realidad virtual que cumple tres requisitos:

- 1. Combinación de elementos virtuales y reales.*
- 2. Interactividad en tiempo real.*
- 3. Información almacenada en 3D. "*

Azuma (1997)

El presente trabajo se enmarca entre ambas definiciones, siguiendo la teoría desarrollada por Paul Milgram y Fumio Kishino en 1994 llamada Milgram-Virtuality Continuum, generando un entorno de realidad virtual semi-inmersiva, en el que interactuamos con el mundo virtual de una forma intuitiva y en tiempo real, pero a través de un monitor.

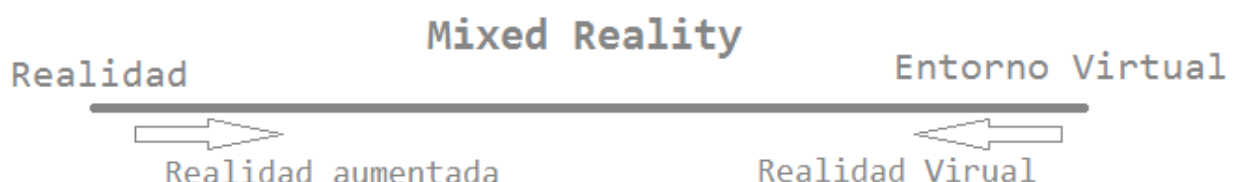


Figura 1: Virtual-Reality Continuum

Es importante aclarar que cuando se habla de Sistemas de Tiempo Real pueden ser tanto sistemas “Hard” construidos sobre sistemas operativos de tiempo real en los que las deadlines son críticas, como “Soft” en los que el tiempo real se refiere a la forma de la interacción de los deadlines perdidos con el rendimiento de la aplicación. Este proyecto es el desarrollo de un sistema del segundo tipo; esto se discute más detenidamente en el apartado 3.2.2- Relación y descripción de los ficheros generados - Justificación uso de sockets UDP/UNIX.

Elementos necesarios para realizar un sistema de realidad aumentada:

- En primer lugar, se necesitan elementos que adquieran la información de la realidad que rodea al sujeto objetivo de la realidad virtual. Para ello se usan tanto cámaras para captar el entorno, como sensores u otras cámaras para captar al propio sujeto, y al usuario del sistema de realidad virtual. Ejemplos típicos son acelerómetros para saber la posición en la que se halla el punto de vista, geolocalización para conocer la posición del usuario, y flexómetros para conocer la pose de los elementos representados.
- En segundo lugar, es necesario un elemento de procesamiento de imágenes que permita la función de todos los datos en un entorno inteligible para el usuario. Para ello son necesarios ordenadores relativamente potentes, aunque como se indicó anteriormente este punto está perfectamente desarrollado en la actualidad con una capacidad suficiente en un ordenador personal medio, e incluso en muchos terminales móviles.
- En tercer lugar, es necesario el elemento sobre el que se proyectarán los resultados del procesamiento de imágenes del apartado anterior. Éste puede ser una pantalla o una superficie genérica como un monitor o un proyector, o específica como los displays de las gafas de realidad virtual y aumentada.

1.1.2 Estado del arte

Aunque en muchas fuentes se considera realidad virtual cualquier forma de interacción con un ordenador, realmente es a principios de los años ochenta cuando se dispone de ordenadores suficientemente potentes como para desarrollar sistemas realmente inmersivos, como el primer simulador de vuelo desarrollado en 1981 por Thomas A. Furness III, llamada “Cabina Virtual”. Es en

los años noventa cuando los fabricantes de videojuegos empezaron a lanzar sistemas realmente inmersivos, y los simuladores profesionales finalmente empezaron a utilizarse masivamente. De esta forma, es apenas hace treinta años cuando la física de experiencias parcial o totalmente virtuales toma forma. A partir de entonces ha sufrido una enorme diversificación.

Actualmente estamos viviendo los primeros pasos en la maduración de la segunda explosión de realidad virtual. La puesta a la venta de dispositivos de realidad virtual para el consumidor medio, así como la generalización de aplicaciones de realidad virtual en la vida diaria y profesional, están generando que tanto las grandes empresas como pequeñas start-ups y universidades estén dedicando esfuerzo y recursos a desarrollar experiencias de realidad virtual para el gran público. Los principales focos de avance de esta segunda era han sido:

- Mejora de la velocidad y el rendimiento de las CPUs y GPUs, lo que permite trabajar con gráficos más ricos y realistas.
- Aumento de la capacidad de las memorias físicas, tanto caché y RAM para la ejecución de los programas como de almacenamiento masivo en las que se pueden guardar archivos gráficos mucho más pesados.
- Mejora de la resolución y calidad de las pantallas, así como de las baterías, dando como resultado dispositivos relativamente pequeños y portables con autonomías de varias horas.

Campos de aplicación

A continuación se expone el estado de la realidad virtual y aumentada por sectores.

Sector del entretenimiento:

Sin lugar a dudas, el auténtico motor de la innovación en realidad virtual en los últimos años está siendo la industria del entretenimiento electrónico. Han surgido gran número de startups y proyectos de empresas ya consagradas dedicados a los productos de realidad virtual como “*Oculus Rift*” desarrollado por Oculus VR, comprada por Facebook en marzo de 2014, o “*Google Cardboard*”, producto creado por Alphabet.

La industria del videojuego se está volcando con la realidad virtual, y esto se puede ver en la gran cantidad de productos, aunque la mayor parte aun en fase de desarrollo, que muestran en las expos de electrónica de consumo.

Sector industrial:

En el entorno industrial, está normalizado el hecho de que la realidad virtual y la realidad aumentada forman parte de la Industria 4.0, ya que estas tecnologías permiten que sus trabajadores reciban, procesen y envíen informaciones relevantes en tiempo real, facilitando el trabajo en equipo y la planificación. Un ejemplo clásico ya consolidado de este tipo de realidad aumentada son los HUDs o “Heads Up Displays” de los cuadros de mandos de los aviones, tanto militares como civiles, que además están siendo progresivamente incluidos en coches de alta gama. Estos HUDs proyectan sobre el cristal frontal de la cabina, o sobre la pantalla del casco del piloto, los datos de control como velocidad, altitud, inclinación alrededor de los ejes etc.

Otra aplicación que próximamente formará parte de la Industria 4.0 son sistemas que se están desarrollando para, por ejemplo, proyectar en tiempo real datos de los distintos elementos de una fábrica en el visor de un casco, de forma que el trabajador puede mirar por ejemplo a una válvula, y conocer su estado, temperatura, nombre etc. Su funcionamiento es similar al de un HUD tradicional, pero requiere una mayor interconexión dentro de la fábrica, un tratamiento de datos muy rápido, así como una localización muy exacta del operario en el entorno.

Sector del diseño:

Muy importante dentro del sector industrial es la ayuda que proporciona la realidad virtual al proceso de diseño. Grandes empresas se están enfocando en el desarrollo de soluciones informáticas de realidad virtual que permitan a diseñadores, ingenieros y arquitectos la visualización y manipulación de sus diseños, fundamentalmente a través de maquetas virtuales, que pueden ser dinámicas para por ejemplo ver los efectos aerodinámicos de una simulación a medida que se desarrolla, o recorrer virtualmente un edificio antes de que se construya pudiendo hacer manipulaciones sobre el mismo.

Ejemplos de esto son el software desarrollado para visualizar un coche antes de su fabricación. A partir de su modelo totalmente ensamblado en 3D en un ordenador, los diseñadores e ingenieros pueden, con un dispositivo móvil, verlo

en el espacio desde cualquier ángulo, ayudando a hacerse una mejor idea del mismo.

Sector médico:

En el sector de la medicina se llevan varias décadas buscando la forma de implementar los avances tecnológicos en telecomunicaciones y realidad virtual. Las operaciones a distancia son un hecho, y sobre ellas se está desarrollando un nuevo campo para aplicar la realidad aumentada para facilitar el trabajo a los cirujanos.

El ejemplo clásico de operaciones con realidad aumentada, y también a distancia, son las operaciones de laparoscopia, que desarrollaremos en el siguiente punto 1.1.3- Introducción a la cirugía laparoscópica y HALS.

1.1.3 Introducción a la cirugía laparoscópica y HALS

La cirugía laparoscópica nace en los años 70 con el objetivo de reducir el tamaño de las incisiones en las operaciones de torso y abdomen. Hasta entonces, el enfoque era el de laparotomía, con una o varias grandes incisiones a través de las cuales los cirujanos acceden al interior de la cavidad abdominal del paciente. La cirugía laparoscópica sustituye este tipo de incisiones por varias pequeñas incisiones realizadas en puntos clave. Cada incisión se denomina "puerto", y en cada puerto se inserta un instrumento tubular conocido como trocar. A través de los trocares se introducen útiles quirúrgicos adaptados para este tipo de operaciones. Para proporcionar al cirujano un espacio de trabajo y visibilidad, al iniciar el procedimiento el abdomen se infla con dióxido de carbono. De esta forma las operaciones son mucho menos invasivas y los tiempos de recuperación y el riesgo de infecciones en las cicatrices se reducen.

Desde la primera laparoscopia en humanos realizada en 1975 por el Dr. Tarasconi se ha expandido esta técnica a gran parte de las operaciones pélvico-abdominales, necesitando normalmente tres incisiones. El cirujano usará dos para introducir sendas herramientas en el cuerpo del paciente, y a través de la tercera un asistente introducirá una cámara que permitirá al cirujano ver la operación en una pantalla.

Si bien la laparoscopia ha significado un avance muy importante en las operaciones pélvico-abdominales, también supone un problema a la hora de

realizar ciertas intervenciones. Éstos son fundamentalmente dos:

- Al no introducir las manos dentro del paciente, el cirujano pierde totalmente el sentido del tacto.
- Al no realizarse incisiones grandes no se pueden extirpar órganos en caso de ser necesario, como en las en operaciones de esplenectomía (extirpación de bazo) y en colectomías (extirpaciones de colon).

A estos hay que sumar una lógica limitación en la visión del facultativo.

Operaciones HALS – Una solución intermedia

Es para hacer frente a estos problemas por lo que nace la técnica para la que se está desarrollando el proyecto al que pertenece este TFG. Las operaciones HALS (“Hand Assisted Laparoscopic Surgery”) nació en los años noventa para devolver la sensibilidad de una mano al cirujano, y permitir la extracción de órganos, conservando en la medida de lo posible la filosofía de la laparoscopia de minimizar las incisiones y la invasividad de la operación.

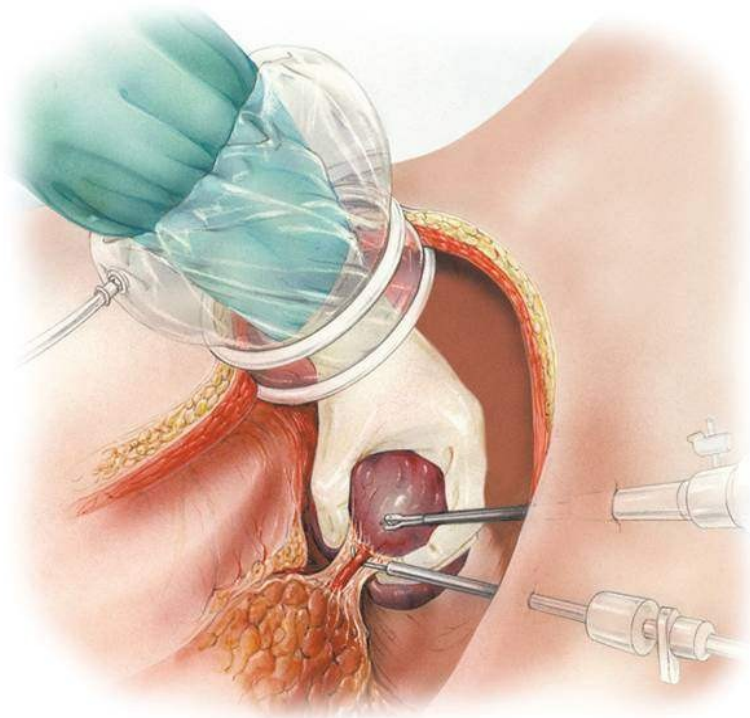


Figura 2: Imagen de una operación HALS. Cortesía del World Laparoscopic Hospital

Este tipo de intervenciones consisten en una operación de laparoscopia tradicional, pero el cirujano introduce la mano no dominante en la cavidad abdominal del paciente a través de una incisión más grande que las otras, de forma que pueda utilizar esa mano para la retracción de órganos en sus dos acepciones, apartarlos para mejorar la visibilidad o facilitar el acceso, y la retirada permanente de una porción o totalidad del volumen. Además, en situaciones críticas facilita el control de sangrados.

Actualmente la cirugía HALS es aplicada fundamentalmente en operaciones que, debido a su propia naturaleza, no se puede determinar si con una operación laparoscópica bastaría, o si en cambio será necesario hacer una laparotomía tradicional.

1.1.4 Marco del proyecto: Sistema robotizado colaborativo para cirugía laparoscópica asistida por la mano

Este Trabajo de Fin de Grado se enmarca dentro de un proyecto coordinado que busca el desarrollo de un entorno integrado robotizado colaborativo para cirugía laparoscópica asistida por la mano.

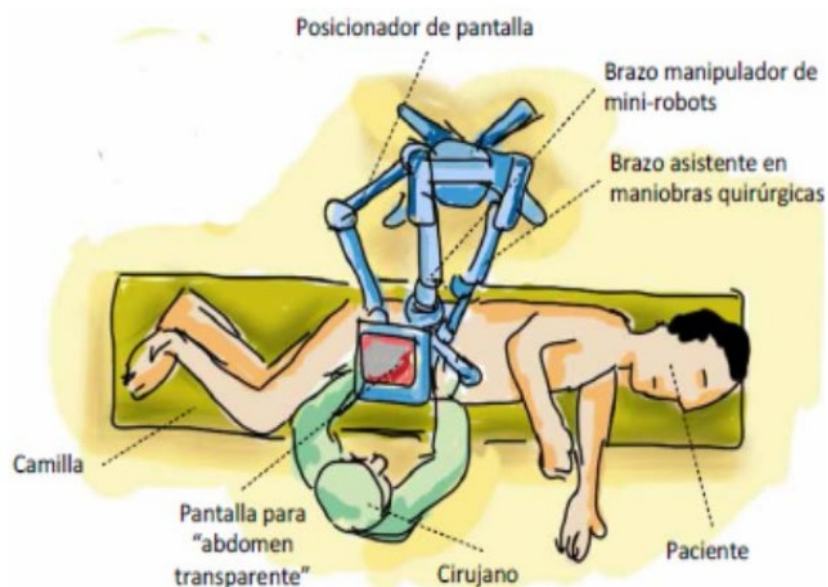


Figura 3: Operación en entorno asistido por Robot. Cortesía de Víctor Muñoz

El sistema consta de un robot colaborativo capaz de manejar un endoscopio y una herramienta laparoscópica articulada, y un sistema especializado en el movimiento de minirobots dentro de la cavidad abdominal.

Por otro lado, se incluirá un interfaz persona máquina que busque emular el concepto de “abdomen transparente”, a partir de los datos obtenidos por la cámara endoscópica manejada por el robot, y los datos aportados por un guante quirúrgico sensorizado, mediante un entorno de realidad aumentada.

El sistema recogerá los datos del gesto de la mano,, los movimientos de los instrumentos en el interior del abdomen, y la lectura de señales fisiológicas combinadas con un modelo paciente-intervención, para crear un entorno de cirugía asistida integrado.



Figura 4: Fotografía del entorno de desarrollo

Dentro de este proyecto, el presente TFG trabaja en el desarrollo de la interfaz persona máquina para dar una base sobre la que desarrollar el abdomen transparente. El objetivo, que se detalla en la siguiente sección, es la creación de una capa de comunicación desde el guante sensorizado hasta un interfaz gráfico así como el desarrollo del propio interfaz.

1.2 Objetivos

1.2.1 Objetivo global

El objetivo de este TFG es el desarrollo de un interfaz gráfico de usuario para un guante sensorizado, lo que incluye también la capa de comunicación entre el guante y el entorno de realidad virtual. Además, el proyecto incluye el requisito de que la transmisión de los datos desde el guante hasta el interfaz pase a través del programa ROS (“Robotic Operating System”) para la publicación de los datos, de forma que éstos sean públicos para el resto de partes del proyecto global que se introdujeron en la sección anterior 1.1- Marco del Proyecto. El funcionamiento de ROS se detalla en la sección 2.1 Planteamiento inicial del proyecto.

Debido a la complejidad del proyecto, se ha optado por dividirlo en cuatro bloques o hitos para un mejor desarrollo del mismo, así como una mejor comprensión del trabajo realizado.

1.2.2 Objetivos parciales

Hito 1 – Recolección y publicación de los datos necesarios

El primer hito consiste en la adaptación de dos programas ya creados por mi 2ª Tutora Dña. Lidia Santos para la lectura de los datos desde el guante y su posterior publicación en ROS. Esta parte está desarrollada en C++ por especificaciones del proyecto.

Hito 2 - Desarrollo de un programa puente para la conexión a un entorno de realidad virtual

En este segundo hito se crea una librería que facilite el acceso desde un entorno de realidad virtual a los datos publicados por ROS, así como dos programas totalmente funcionales que realicen esta labor. Uno accede a y transmite los datos de funcionamiento del guante, y otro accede a los datos de calibración del mismo (ambos tipos son publicados en ROS). Este apartado se ha desarrollado en Python por razones que se exponen más adelante en el apartado 2.2- Modificaciones.

Hito 3 – Desarrollo de una mano virtual

El tercer hito es el desarrollo de una mano virtual en un entorno de realidad virtual que reproduce los movimientos de una mano real que utilice el guante sensorizado. Este apartado se ha realizado en el programa Blender, con la programación desarrollada en Python. La justificación está de nuevo 2.2-Modificaciones.

Hito 4 – Desarrollo de un interfaz de usuario para integración del proyecto

Finalmente, se ha creado un GUI en forma de botonera que permite compilar y lanzar los programas detallados en los hitos anteriores. Este cuarto hito no estaba inicialmente planeado, pero ha sido desarrollado debido a la dificultad en el setup de todo el proyecto. Las razones se detallan de nuevo 2.2-Modificaciones.

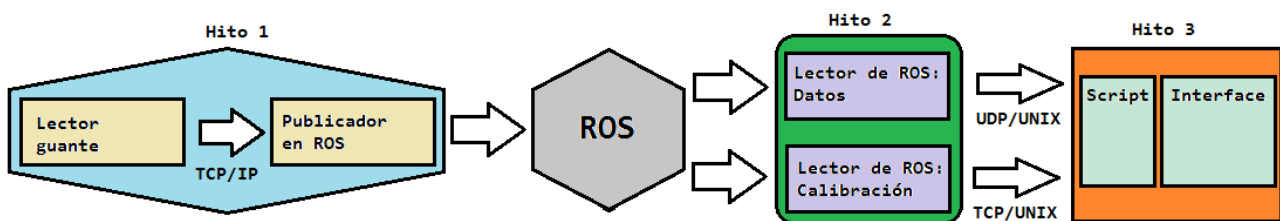


Figura 5: Esquema general de la aplicación

1.3 Planteamiento del TFG

Como se apuntó 1.2- Objetivos, el proyecto ha sufrido algunas variaciones de forma desde su inicio, si bien los objetivos han permanecido inalterados.

El planteamiento inicial era a partir de los datos proporcionados por el guante sensorizado *5DT Data Glove 14 Ultra (Left)*, recolectados y transmitidos a ROS por los programas de mi 2ª Tutora Dña. Lidia Santos, leerlos y transmitirlos al entorno Gazebo, en el que se habría modificado la simulación Shadowhand

para su uso con este guante. A continuación se detallan todas las partes de las que se compone, las razones por las que se buscó una alternativa, y el funcionamiento final.

1.3.1 Materiales utilizados y descartados

Sistema Operativo

Como sistema operativo se ha utilizado Ubuntu 14.04 LTS. Se trata de una distribución de escritorio GNU/Linux libre basado en Debian para ordenadores personales. Está publicado y mantenido por Canonical Ltd. Hasta el 04/2019.

Se ha utilizado este sistema operativo por la compatibilidad con ROS Índigo, la gran comunidad online que tiene debido a su gran cuota de mercado de las distribuciones GNU/Linux (más del 50% en ordenadores personales), y el margen que tiene de soporte por parte de la empresa.

Guante sensorizado

Dentro de las especificaciones del TFG está el uso del *5DT Data Glove 14 Ultra (Left)*. Los estudios comerciales comparativos se han realizado previamente y escapan a este trabajo.

El seleccionado es un guante con catorce sensores ópticos de flexión, dos por dedo para flexión longitudinal y uno entre cada par de dedos para conocer la posición relativa de uno respecto de los otros. Se corresponde con la mano izquierda, ya que como se dijo 1.1.3- Introducción a la cirugía laparoscópica y HALS, los cirujanos prefieren utilizar su mano no dominante, y la mayoría de la población es diestra. Las especificaciones del guante son las siguientes:

Specifications		
	5DT Data Glove 5 Ultra	5DT Data Glove 14 Ultra
Material	Black Stretch Lycra	Black Stretch Lycra
Sensor Resolution	12-bit A/D (typical range 10 bits)	12-bit A/D (typical range 10 bits)
Flexure Sensors	Fiber Optics Based 5 Sensors in total 1 Sensor per finger, measures average of knuckle and first joint.	Fiber Optics Based 14 Sensors in total 2 Sensors per finger, one sensor for knuckle and one for first joint. Abduction sensors between fingers.
Computer Interface	Full-speed USB 1.1 RS-232 (via optional serial interface kit)	Full-speed USB 1.1 RS-232 (via optional serial interface kit)
Power Supply	Via USB Interface	Via USB Interface
Sampling Rate	Minimum 75Hz	Minimum 75Hz

Figura 6: Tabla de especificaciones para guantes 5 Ultra y 14 Ultra

Este guante viene acompañado de la librería “fglove.h” proporcionada por el fabricante. Básicamente es una API para leer la comunicación serial que suministra el guante, a través de ella podemos abrir el puerto y leer del mismo los datos que proporcionan los sensores con funciones de alto nivel.

ROS

La definición que dan de sí mismos es "ROS (Robot Operating System) proporciona librerías y herramientas para ayudar a los desarrolladores de software a crear aplicaciones robóticas. Proporciona abstracción del hardware, drivers para dispositivos, librerías, visualizadores, paso de mensajes, gestión de paquetes y más". ROS está licenciado bajo una licencia Open Source BSD.



Figura 7: Estructura de ROS. Cortesía de [Open Source Robotics Foundation](http://www.open-source-robotics.org/)

Dentro de las enormes capacidades que ofrece ROS, nosotros haremos uso de la nuclear: el paso de mensajes. Al nivel más bajo, ROS proporciona un interfaz para la publicación/subscripción mensajes anónimamente. A continuación se desarrolla esta funcionalidad y el paradigma que implementa.

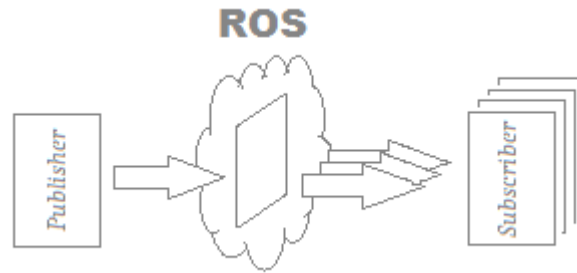


Figura 8: Función de ROS en el paradigma Publicador/Subscriber

ROS como entorno para Publicación/Subscripción anónima:

Usaremos ROS para implementar el paradigma Publicador/Subscriber a través de su núcleo (roscore). Las partes de las que se compone son las siguientes.

- Roscore: es el núcleo de ROS. Se debe lanzar para permitir la comunicación entre los publicadores y los subscriptores. Es el entrono central al que llegan para publicarse y del que se leen los mensajes creados por los publicadores.
- Publicadores: son procesos que comunican datos de forma que cualquier proceso con la clave de la publicación puede acceder a ellos. En ROS los publicadores hacen uso de dos claves:
 - El “Topic”: es un bus con nombre sobre el que se van a publicar los mensajes, en el que se engloban los distintos chats en los que se publican los datos. Los topics tienen semánticas anónimas para la publicación/subscripción, lo que quiere decir que un proceso publicador no sabe qué procesos están leyendo de su topic, ni los subscriptores conocen el proceso publicador.
 - El “Chat”: Dentro de cada “topic” puede haber uno o varios chats. Es el identificador con el que se reconoce un entorno de publicación. Un subscriptor se puede conectar a uno de estos chats para leer el dato que esté ahí publicado.
 - El paradigma Publicador/Subscriber: la idea detrás del paradigma publicador/subscriptor es la de poder hacer pública una información relevante para algunos procesos, de forma que sean capaces de leer la misma información publicada una sola vez varios procesos sin

consumirla.

Un ejemplo físico sería un tablón de anuncios de casas. Si nos interesa comprar una casa, podremos acceder para obtener información en un portal especializado, pero que nosotros leamos un anuncio no implica que éste desaparezca, sino que permanecerá disponible hasta que lo reemplace un anuncio más actualizado.

- **Subscriptores:** son el final de la comunicación. Utilizan el nombre del “chat” para conectarse a la conversación y leer la información publicada. Puede haber tantos Subscriptores como deseemos conectados a un chat, sin que la información desaparezca.

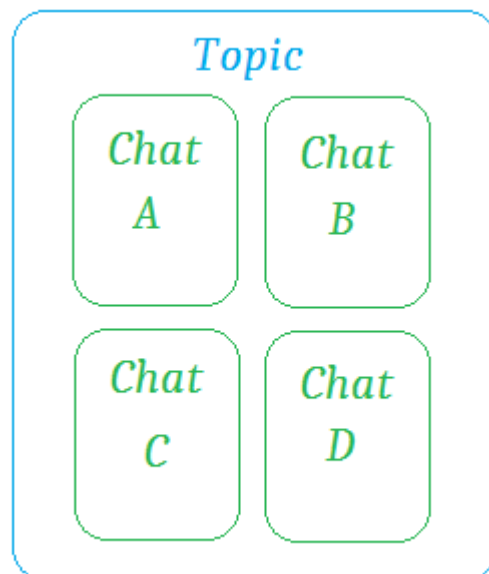


Figura 9: Estructura Topic-Chats

Catkin

Catkin es el “build system” oficial de ROS y sucesor del build system original rosbld. Un “build system” es responsable de generar targets de código fuente para ser compilados y que puedan ser usados por un usuario final o “end user”. Estos targets pueden ser librerías, ejecutables, scripts, interfaces exportados (cabeceras C++) o cualquier otra cosa que no sea código estático. Ejemplos de estos son GNU Make, Cmake (el utilizado por Catkin), o Apache Ant. Catkin combina los macros de Cmake con scripts en Python para proporcionar funcionalidad sobre el flujo de trabajo habitual de CMake.

En ROS, el código fuente está organizado en paquetes, donde en cada paquete suele haber más de un target para construir. Estos targets se definen en el documento CMakeLists.txt en el que se especifican las rutas, las dependencias y los argumentos de cada uno, de forma que para construirlo sencillamente introduzcamos el comando 'catkin_make' y él solo construya todo el paquete.

Catkin fue diseñado para ser más convencional que rosbuilt, permitiendo una mejor distribución de paquetes, mejor soporte coss-compiling y mejor portabilidad. Básicamente catkin funciona igual que CMake, pero añade una capa para encontrar distintos paquetes, y construirlos a la vez.

Gazebo y Shadowhand

Gazebo es un simulador 3D de alta fidelidad capaz de representar poblaciones de robots en entornos complejos de interior y exterior. Shadow hand es una simulación de una mano robótica que existe en la realidad.



Figura 10: Shadowhand

La idea original del proyecto era modificar el código fuente de shadowhand para adaptarlo a las necesidades del proyecto general descrito en el apartado 1.1.4- Marco del proyecto, pero tras estudiarlo detenidamente un tiempo se vio que iba a resultar más costoso, con un resultado peor, que crear un modelo nuevo desde cero. Las razones de más peso son las siguientes:

- El modelo de la mano se corresponde con una mano robótica, no con una mano real.
- Siguiendo esto, la mano robótica no dispone de las articulaciones de rotación en los nudillos que se necesitan para nuestro proyecto.
- Además, el modelo está importado directamente a gazebo, por lo que no se podía modificar más que a nivel de código.
- Está en parte programado en Lisp, lenguaje del que no se tiene conocimiento.
- Está desarrollado para ROS fuerte, que solo es compatible con Ubuntu 12.04 LTS por lo que habría que migrarlo a Índigo.
- Muchas partes del código están diseñadas específicamente para la mano robótica: sensores de fuerza, de velocidad, de posición, táctiles etc.

Por todo lo anterior, quedó claro que modificarlo para adaptarlo a nuestros propósitos iba a dar un resultado bastante mediocre, o incluso que fuera imposible de adaptar al final, por lo que se inició una búsqueda de alternativas intensiva, en la que tras probar distintas posibilidades se llegó al siguiente planteamiento.

1.3.2 Modificaciones

Rospy

ROS dispone de una librería de cliente para Python 2.6. Tras estudiar los servicios que presta, comprobamos que se trata de una API muy completa, que permite hacer los mismos desarrollos que el C++, pero de una forma más sencilla, debido al más alto nivel de programación que supone Python. Como el uso de rospy no afecta a la funcionalidad que se busca desarrollar, no ha

habido inconveniente en el desarrollo del Hito 2 con esta API en lugar de la API tradicional de C++ usada en el Hito 1.

Blender

Blender es una suite de creación 3D libre y de código abierto, mantenida por la Blender Foundation. Permite trabajar sobre todo el proceso de creación 3D: modelado, “rigging” (articulado), animación, simulación, renderizado, composición y seguimiento de movimientos. Además, cuenta con su propio editor de vídeo para las animaciones, con su propio motor de videojuegos, y con una API para Python (con su propio núcleo de Python 3.5) que permite editar la aplicación y escribir scripts para su ejecución dentro de las animaciones o del motor de videojuegos.



Figura 11: Portada de Blender 2.78a, la versión utilizada

Blender es cross-platform, es decir funciona igual e igual de bien en Linux, Windows y Macintosh (no tiene desarrollo asimétrico por plataforma). Al tener su propio núcleo de Python permite la portabilidad de los archivos, y utiliza OpenGL para proporcionar una experiencia consistente.

Al estar bajo licencia GPLv2 o superior (dependiendo de la versión) el usuario es libre de usar Blender para cualquier propósito, incluyendo comerciales o de educación. Esto además hace que disponga de una comunidad muy amplia, con sección propia en StackExchange y foros especializados.

Como se puede intuir, la compactación de todo el proceso de desarrollo en un

solo entorno, así como la API de Python y el motor de videojuegos que nos permite interactuar con la escena a través de scripts lo hicieron muy atractivo para esta aplicación. Tras un estudio preliminar, se acordó con los tutores de este TFG el uso de Blender en sustitución de Gazebo y Shadowhand para la construcción del Hito 3.

GUI con Tkinter

Hacia el final del proyecto se vio la necesidad de incluir un cuarto hito, el desarrollo de una interfaz gráfica de usuario para la gestión del lanzamiento y apagado de la aplicación, debido a su gran volumen.

Se decidió implementar en Python 3, utilizando una librería llamada Tkinter que se discutirá en el apartado 3.4- Hito 4.

2 Desarrollo del TFG

2.1 Hito 1

En este primer hito se ha desarrollado la capa de comunicación entre ROS y el guante sensorizado. El objetivo es publicar en ROS los datos necesarios para la el funcionamiento de la mano virtual. Como se indicó 1.2.2- Objetivos parciales - Hito 1 los programas de base ya están creados, de forma que la tarea es adaptarlos para la funcionalidad requerida, por lo que solo me explayaré en aquellas partes desarrolladas expresamente en este TFG, pasando someramente por las funcionalidades preexistentes que sean necesarias para la comprensión del funcionamiento de este proyecto.

El primer programa utilizado es `dglove.cpp`. Se trata de una modificación del programa `glove.cpp` desarrollado por Dña. Lidia Santos, adaptándolo a las necesidades del TFG.

Este programa utiliza la API que proporciona 5DT para la gestión software del guante sensorizado. A continuación trataremos las funciones utilizadas en el programa. Para más información sobre la librería "`fglove.h`", me remito a la documentación de la misma incluida en la bibliografía.

Recolección de datos

Al estar trabajando en un sistema GNU/Linux, la gestión de cualquier dispositivo se realiza a través de udev como un directorio en la ruta `'/dev'`. En nuestro caso, la ruta utilizada será `"/dev/usb/hiddev0"`. A través de la API accederemos a este dispositivo si otorgamos permisos de administrador al programa.

La configuración del guante es semiautomática, ya que se programa a mano pero a través de funciones ya incluidas en la librería `fglove.h`. Lo primero que se realiza es un chequeo del tipo de guante conectado, de forma que se detectan el número de sensores y la mano a la que se corresponde.

El guante transmite por comunicación serie los valores brutos leídos por los sensores. Como se trata de una conversión analógico/digital, el guante transmitirá un valor en binario de 'n' bits de resolución mapeado a partir del

valor analógico del sensor óptico con una resolución de 2^n valores. En nuestro caso, como el guante que tenemos posee 12 bits de resolución, cada sensor transmitirá un valor sin tratar entre 0 y 4095 ($2^{12}-1$)

$$\text{Resolución} = 2^{\text{número de bits}} = 2^{12} = 4096 \text{ niveles } [0 - 4095]$$

Figura 12: Fórmula de la resolución en bits

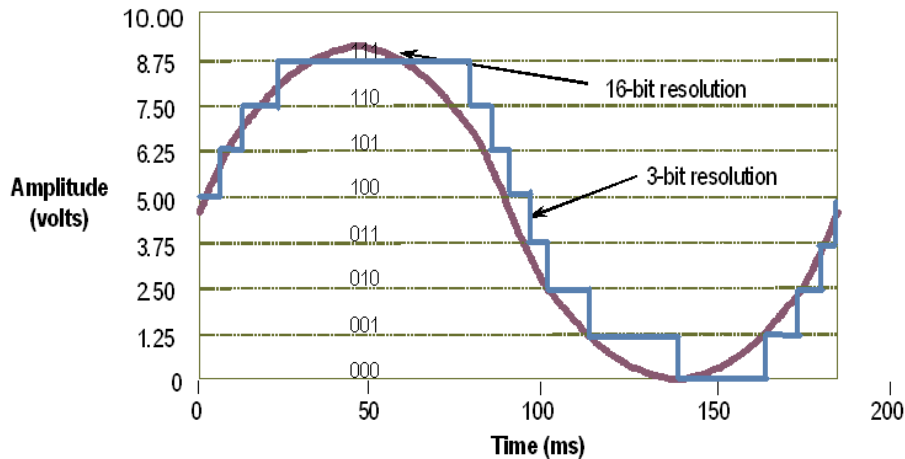


Figura 13: Muestreo digital de una señal analógica

A partir de los datos brutos, la API nos permite convertirlos automáticamente a valores en porcentaje de apertura si previamente hemos realizado una calibración, también por medio de la API. Lo cierto es que este proceso no quedaba claro en la documentación de la librería, por lo que solicité más información a la empresa, que me remitió un manual de calibración, en el que se detalla el siguiente proceso.

Calibración del guante

Para calibrar el guante es necesario colocar la mano en diversos gestos, de forma que se alcancen los puntos de máxima y mínima flexión de cada sensor por separado. En el manual que se remitió por parte del fabricante 5DT especifican un proceso de cuatro pasos con los sensores que se deben recoger en cada gesto:

1- Mano colocada plana sobre una superficie con todos los dedos juntos:

- a. Thumb/Index - Minimum Abduction
- b. Index Near - Minimum Flexure
- c. Index Far - Minimum Flexure
- d. Index/Middle - Minimum Abduction
- e. Middle Near - Minimum Flexure
- f. Middle Far - Minimum Flexure
- g. Middle/Ring - Minimum Abduction
- h. Ring Near - Minimum Flexure
- i. Ring Far - Minimum Flexure
- j. Ring/Little - Minimum Abduction
- k. Little Near - Minimum Flexure
- l. Little Far - Minimum Flexure

2- Mano cerrada con el pulgar hacia arriba

- a. Thumb Near - Minimum Flexure
- b. Thumb Far - Minimum Flexure
- c. Index Near - Maximum Flexure
- d. Index Far - Maximum Flexure
- e. Middle Near - Maximum Flexure
- f. Middle Far - Maximum Flexure
- g. Ring Near - Maximum Flexure
- h. Ring Far - Maximum Flexure

i. Little Near - Maximum Flexure

j. Little Far - Maximum Flexure

3- Mano extendida con los dedos juntos y el pulgar sobre la palma

a. Thumb Near - Maximum Flexure

b. Thumb Far - Maximum Flexure

4- Mano extendida sobre una superficie plana con los dedos separados

a. Thumb/Index - Maximum Abduction

b. Index/Middle - Maximum Abduction

c. Middle/Ring - Maximum Abduction

d. Ring/Little - Maximum Abduction

Una vez recogidos estos datos en dos vectores de 14 unsigned short por especificaciones de la API, se pasan a la función `fdSetCalibrationAll` que escalará todos los sensores, permitiendo el uso de las funciones escaladas de que dispone la librería.

El sistema de calibración del guante utiliza la siguiente fórmula, que da por supuesto que la flexión máxima de cada articulación va a ser de 90°:

$$AngleCurrent = \frac{(DataCurrent - DataMin)}{(DataMax - DataMin)} \times (MaxDeg - MinDeg) + MinDeg$$

Figura 14: Fórmula de la calibración

$$((2700 - 2000) / (3000 - 2000)) \times (90 - 0) + 0 = 700 / 1000 \times 90 = 63^\circ$$

Figura 15: Ejemplo de valores de ángulo

He tratado el tema con mis tutores, y si bien es verdad que hay articulaciones, en particular las falanges medias, que son capaces de doblarse más de 90°, debido a la flexibilidad del tejido del guante y la propia rigidez del sensor, el ángulo medido por éste no varía tanto como se podría esperar, por lo que

hemos considerado que estos 90º grados son una aproximación suficientemente buena. Por otro lado, los datos experimentales que dan los sensores interdigitales son bastante malos.

Hasta aquí la parte de recolección de datos, a continuación se expone cómo estos datos son transmitidos hasta ser publicados en ROS.

Transmisión de los datos

Por requerimientos del proyecto, es posible que el programa que lee del guante y el núcleo de ROS (`roscore`) estén corriendo en máquinas distintas, separadas físicamente. En la estructura original de esta parte de la comunicación se había optado por la separación del programa gestor del guante y el publicador en ROS, que se comunicarán a través de sockets TCP/IP. En nuestro caso se ha especificado como dirección IP 127.0.0.1 (localhost), pero si se separasen, adaptarlo sería sencillamente cambiar esta dirección IP. El socket tendrá un BACKLOG de 1 ya que solo se conectará a él un proceso que recogerá los datos y los publicará en ROS.

Los datos a transmitir en principio eran solo los datos que está continuamente el guante proporcionando desde los sensores. Sin embargo, se pensó que si se desea trabajar con los datos bastos, o realizar algún tipo de ajuste fino de la calibración, sería necesario transmitir los datos de esta. Por ello, se ha establecido un pequeño protocolo sobre los sockets para indicar si los datos comunicados se corresponden a la calibración o al normal funcionamiento.

Protocolo

El paquete está conformado con una cabecera de 1 byte (8 bits) en la que se indica el tipo de dato a transmitir, por medio de un carácter:

“01000100” → ASCII 'C' → Calibración

“01000101” → ASCII 'D' → Datos funcionamiento

y por un campo de datos de 56 bytes (448 bits) que se corresponde con los 14 float.

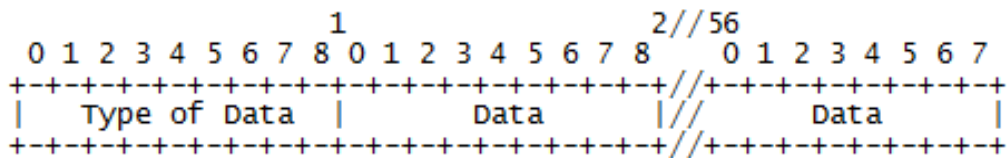


Figura 16: Estructura del paquete del protocolo

Este protocolo no tiene handshake ni ACK. La comunicación es directa y no confiable, ya que se descarga esa responsabilidad en el protocolo TCP/IP que subyace. Una vez establecida la comunicación el primer paquete que se envía es la calibración máxima, segundo la calibración mínima, y a continuación se envían los paquetes de funcionamiento normal de manera continua. El cliente debe estar preparado para tratar los paquetes sin que se haya solicitado nada más que una comunicación TCP/IP.

La comunicación es en primer lugar tomada por el protocolo TCP que realiza un handshake enviando un paquete con el flag SYN desde el cliente al servidor. Este responde con un paquete con los flags SYN y ACK, a lo que el cliente responde con un paquete con el flag ACK. Una vez realizado esto, queda establecida la conexión y el servidor, sin esperar una petición ya que considera como petición el establecimiento de la conexión, manda los dos paquetes de la calibración máxima y mínima. A continuación empieza a enviar los valores leídos del guante.

La comunicación termina cuando el cliente envía un paquete con el flag FIN. A esto el servidor responderá enviando un paquete con el flag ACK para indicar que ha recibido la solicitud, y otro paquete con el flag FIN cuando sea posible cerrar la conexión. Finalmente el cliente envía un ACK confirmando que la conexión se va a cerrar.

Todo este proceso comunicativo puede verse en la figura 3.6 en la siguiente página.

Debido a la adición de este paquete, se ha añadido al programa original una gestión de memoria para la correcta transmisión de los datos en forma de estructura. Para ello se reserva memoria del tamaño de los paquetes usando la función malloc, y se trabaja con punteros a la estructura de memoria reservada. El objetivo de esto es serializar los datos que se quieren enviar, ya que sin serializar no se transmitirán correctamente.

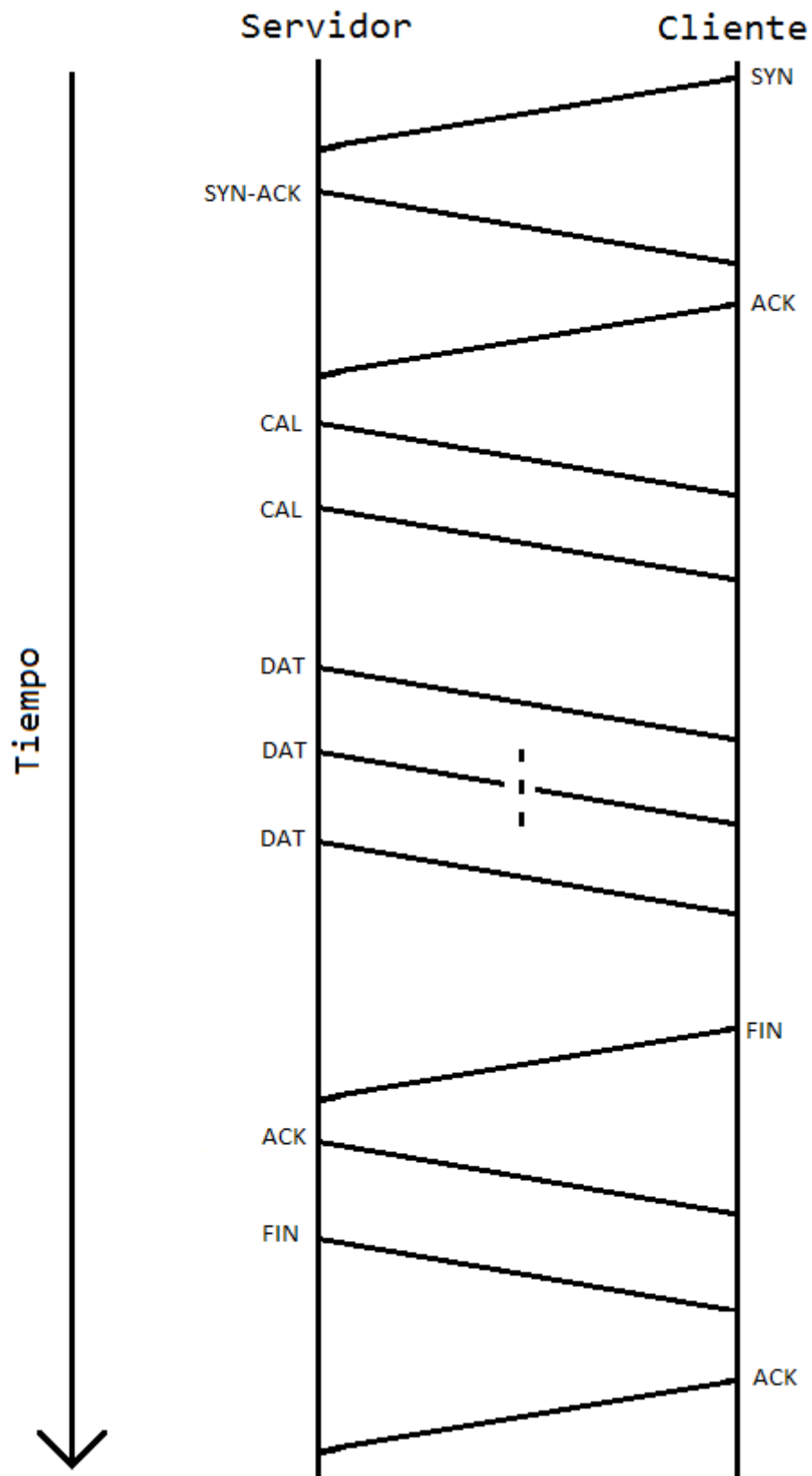


Figura 17: Esquema temporal de la comunicación en el Hito 1

Serialización

La serialización es el proceso de transformar estructuras de datos y objetos en un formato que pueda ser almacenado y transmitido correctamente, de manera que pueda ser recuperado íntegramente en su forma original. Esto se consigue eliminando las relaciones internas entre los bytes, reduciéndolo a un simple “stream” de bytes que permita su tratamiento directo.

La serialización en C es muy compleja, por lo que en este caso lo que se ha hecho es reservar un espacio de memoria sobre el que se escribe directamente el objeto, por lo que la estructura queda fijada. Realmente los datos han sido serializados, pero puede plantear los siguientes problemas:

- “Endianness”: es el orden secuencial usado para interpretar un rango de bytes en memoria como una palabra compuesta de varios bytes. Es el ejemplo de cualquier tipo de variable que no sea 'char' o 'byte'. Esta forma de interpretación varía entre arquitecturas y sistemas operativos. Es por ello que el sistema de serialización usado en esta parte solo funcionará si se usa el mismo Sistema Operativo con la misma arquitectura. En otro caso no se puede garantizar su funcionamiento.
- “Compiler padding”: los compiladores añaden bytes “blancos” entre los bloques de la estructura. Si se utilizan diferentes compiladores en los dos extremos de la comunicación, es posible que el proceso cliente lea algunos de los espacios en blanco, y a cambio no lea datos que sí son relevantes.

Como se ha dicho, esto no representa problema de cara al presente proyecto, porque todo se compila y ejecuta bajo las mismas premisas; sin embargo sería una buena práctica implementar algún tipo de serialización a los programas escritos en C++.

Para facilitar el proceso de serialización han surgido gran cantidad de librerías y estándares. Uno de los más usados actualmente es JSON, cuyo funcionamiento se detalla en el apartado 2.2.2- Relación y descripción de los ficheros generados - JSON.

Finalmente, otra variación que se ha introducido es la adición de una función unificadora de tratamiento de errores, para reducir el código y unificar los códigos de error. Así los siguientes valores se corresponden con las siguientes tipologías de error:

1. ERROR DURANTE LA CALIBRACIÓN
2. ERROR DURANTE LA TRANSMISIÓN DE LA CALIBRACIÓN
3. ERROR DURANTE LA TRANSMISIÓN DE LOS DATOS

Publicador en ROS

El programa `mano.cpp` funciona como cliente de `dglove.cpp` y como Publisher en ROS. Si bien su estructura general ya estaba creada, he realizado una serie de modificaciones para un mejor funcionamiento del mismo, así como su adaptación a las necesidades concretas de mi proyecto.

Una de las necesidades más claras era el poder identificar los datos pasados por los sensores interdigitales de forma que pudiéramos identificar aquellos que hubieran sido leídos a la vez. Esto es necesario ya que la posición que nos indican estos sensores es relativa entre ellos; conocer uno de los valores en un instante de tiempo no nos ayuda a identificar la posición de la mano.

Para solucionar esto, debido a la naturaleza asíncrona de ROS, se ha optado por publicar los datos de los cuatro sensores interdigitales en un solo array con el tipo `Float64MultiArray`. Para ello se ha creado otro Publisher de este tipo, con un tipo de mensaje en el que se cargan los datos que han sido recibidos en un mismo paquete y se envían todos juntos.

La misma solución se ha usado para el caso de la calibración. Se han habilitado dos publicadores más, uno con los datos máximos y otro con los mínimos, que envían mensajes de tipo `Float64MultiArray` con un tamaño de catorce elementos.

En este programa también se ha añadido una función error para unificar los errores producidos y reducir el código similar a la descrita para el programa anterior.

A continuación se explica el proceso seguido para la creación del paquete de ROS, su compilación y su puesta en marcha, necesario para que este último programa publique los datos del guante.

Configuración del paquete

En este apartado describiremos el proceso seguido para crear un paquete de ROS adaptado a este TFG. Dada la calidad de los tutoriales de este programa, se ha considerado innecesario recoger el proceso de instalación de ROS, así que supondremos que partimos de un sistema con ROS Índigo ya instalado y configurado. Para más información, me remito a la wiki de ROS, incluida en la Bibliografía.

Lo ideal y la forma en que vamos a trabajar es creando un paquete de catkin. Como se explicó 2.1.1- Materiales del planteamiento inicial - Catkin, este es un builder para ROS basado en CMake, por lo que parte de la configuración de este paquete se corresponderá con la configuración para CMake.

Pasos a seguir:

- Creamos un workspace de catkin

```
$ cd ~/catkin_ws/src
```

- Usamos 'catkin_create_pkg' para crear la estructura de un paquete. Como dependencias usamos 'std_msgs' para los paquetes del Publisher, 'roscpp' para compilar el Publisher, y 'rospy' para los subscriptores creados en el Hito 2

```
$ catkin_create_pkg tfg std_msgs rospy roscpp
```

- A continuación cambiamos a la raíz del workspace y lo construimos por primera vez, para que lo añada al entorno de ROS

```
$ cd ~/catkin_ws && catkin_make
```

- Copiamos el archivo package.xml proporcionado junto a este documento en la ruta '~/catkin_ws/src/tfg' sobrescribiendo el preexistente. Este archivo es meramente informativo sobre el paquete.
- Copiamos el archivo CMakeLists.txt proporcionado junto a este documento en la ruta '~/catkin_ws/src/tfg' sobrescribiendo el paquete preexistente. Este archivo nos sirve para indicar los objetivos a 'catkin_make' para compilar, así como las dependencias que son necesarias.

- Colocamos en `'~/catkin_ws/src/tfg/src'` los dos archivos de C++.
- Creamos un directorio en `'~/catkin_ws/src/tfg'` con el nombre "scripts". Este es la carpeta donde depositaremos los scripts de Python y bash.

Ahora podríamos compilar por partes el paquete, ejecutando el comando `'catkin_make'` en la raíz del workspace y compilando por otro lado `dglove.cpp` manualmente, pero esto está automatizado desde la interfaz gráfica desarrollada en el Hito 4.

2.2 Hito 2

Como se indicó en la sección 1.2.2- Objetivos Parciales - Hito 2, este hito consiste en la creación de un proceso que lea los datos publicados en ROS por los procesos del Hito 1.

Este hito ha sido desarrollado íntegramente en Python, debido a la facilidad que da la librería `rospy` para desarrollar Publicadores y Subscriptores, también llamados "Talkers" y "Listeners" en el contexto de `rospy`.

En este hito se han desarrollado tres scripts en Python.

- Una librería llamada `listenerGeneric.py` que contiene las clases y métodos utilizados por los siguientes scripts.
- Un script para lanzar los listeners que haciendo uso de la librería leen de ROS y escriben sockets UDP/UNIX, a través de los cuales se envían los datos de funcionamiento al entorno de realidad virtual.
- Un script para lanzar los listeners que leen de ROS y escriben en sockets TCP/UNIX para comunicar los datos de la calibración.

2.2.1 Justificación del interfaz ROS-Entorno virtual

Como se explicó en 2.1.1- Materiales del planteamiento inicial - ROS, un Listener es un proceso que se conecta a un chat de ROS del que lee un dato. A diferencia de los Publishers, que los podíamos lanzar todos desde el mismo proceso, cada Listener se puede conectar a un solo chat, por lo que no

podemos hacerlo como en el Hito 1, sino que usaremos la librería multiprocessing de Python para hacer un spawning desde el proceso padre, y así crear una colección de procesos hijos, cada uno con un listener en su código. Este procedimiento se desarrolla más extensamente en 3.2.2- Relación y descripción de los ficheros generados - Creación de un objeto.

A raíz de esto, quedó clara la necesidad de separar este programa del entorno de realidad virtual. Blender soporta multihilo, pero no multiproceso. Por ello, la primera aproximación fue separar cada listener en un hilo independiente. Sin embargo, las librerías de ROS utilizan interrupciones para indicar a los listeners cuándo leer, en concreto la señal SIGTERM, que solo funciona en el hilo principal:

```
File "/opt/ros/indigo/lib/python2.7/dist-packages/rospy/core.py", line 452, in
register_signals_signalChain[signal.SIGTERM] =
signal.signal(signal.SIGTERM, _ros_signal)

ValueError: signal only works in main thread
```

Esto dio lugar como hemos dicho al script `main_sub.py`. Sin embargo lo que en principio parecía una desventaja al aumentar el número de archivos y etapas de comunicación, se ha sabido aprovechar para añadir funcionalidad y abstracción.

Al separar el proceso de lectura de ROS de la escritura de los datos en el entorno virtual, se ha creado un interfaz entre ROS y este, de forma que cualquier desarrollador que decida implementar una mano virtual en otro entorno de realidad virtual, es decir sustituyendo Blender por otro programa final, puede programar un cliente que se conecte a los sockets que se han implementado en la librería. Siguiendo las directrices marcadas en el Anexo I - Manual de Usuario puede recoger los datos sin tener que implementar la librería de ROS dentro del entorno de realidad virtual, ni tener que compilar el entorno dentro de un workspace de catkin.

2.2.2 Relación y descripción de los ficheros generados

Biblioteca `listenerGeneric.py`

El archivo `listenerGeneric.py` está estructurado de la siguiente manera:

- *Importación de las librerías necesarias:* en Python la clásica instrucción de preprocesador de C '#include' se implementa con la instrucción 'import'. Esta instrucción nos permite importar librerías, pero también partes de ellas con la palabra reservada 'from', así como nombrarlas con alias con la instrucción 'as'.
- *Variables globales:* en Python no existen las constantes como tal, sino que se confía en el buen hacer del programador. Se han declarado tres variables globales, dado su uso extendido en gran parte de las casuísticas del hito.
 - 'address': ruta en la que se crearán todos los sockets '/tmp/'
 - 'nl_fingers': Lista con los nombres de los chats de ROS para los sensores de flexión de los dedos
 - 'nl_sep': Lista con los nombres de los chats de ROS para los sensores de separación interdigitales
- Clases para la creación de los listeners:
 - 'listenerGeneric': Contiene el método `__init__`, que es necesario para inicializar los atributos del objeto, y el método `.run()`, que es convencional como el método por el que empieza su ejecución el objeto. Son iguales en los tres tipos de listeners que tenemos. Se trata por tanto de una clase padre, de la que las siguientes dos heredarán.
 - 'listenerFingers': Clase que hereda `listenerGeneric`, usada para crear listeners para los sensores de flexión longitudinal de los dedos, con una función callback para hacer un tratamiento de los datos recogidos, que en este caso es sencillamente transmitirlos a través de los sockets a Blender.
 - 'listenerExp': Clase que hereda de `listenerGeneric`, usada para crear nodos listeners de ROS para recoger el vector de datos de los sensores de expansión de la mano y su transmisión a través de sockets al entorno de realidad virtual, en nuestro caso Blender.
 - 'listenerCalibration': esta clase, que también hereda de `listenerGeneric`, sirve para crear dos nodos listener de ROS que lean los datos de la calibración y los transmiten al entorno de realidad virtual. El proceso es bastante más complejo que los anteriores, y se

explicará en la siguiente sección.

- Métodos “Lanzadores”: Son tres métodos creados para lanzar instancias de las clases definidas en el punto anterior. Hacen la configuración necesaria de sockets, crean una instancia del objeto y lo lanzan a partir del método `.run()` definido en `listenerGeneric`.

Creación de un objeto

La programación orientada a objetos (OOP) es un paradigma de programación en la que las acciones y la lógica de la programación funcional clásica se ven sustituidos por objetos y datos.

El código se organiza en clases, que contienen una serie de propiedades definidas como variables propias (llamadas atributos) y funciones (llamadas métodos). Estas clases pueden heredar y sobrescribir métodos y atributos de otras clases, como es nuestro caso con la clase padre `listenerGeneric()` y las tres clases hijas correspondientes a los tres tipos de listeners.

Dentro de los lenguajes de programación OOP podemos distinguir dos tipos fundamentalmente:

- 1- Pueden mezclar programación estructurada y OOP, como es el caso de Python.
- 2- Todo debe estar contenido en clases, como es el caso de Java.

Python realmente trabaja solo con objetos, pero a un nivel superior nos permite simular el trabajar en ambos paradigmas (funcional y OOP) a la vez.

El flujo de trabajo que seguimos en los scripts al crear un Listener será el siguiente:

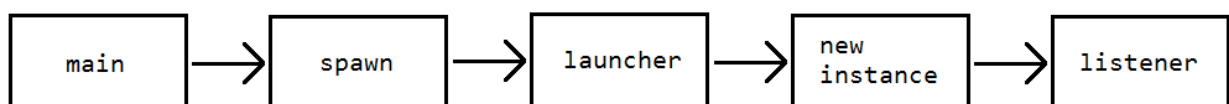


Figura 18: Proceso de funcionamiento del lanzamiento de Listeners

En Python, la librería que permite hacer forks, esto es crear subprocessos hijo a partir del proceso padre, es la librería multiprocessing. Sin embargo, realmente lo que realiza no es un fork como lo entendemos en C, en el que el proceso hijo es una copia del padre, sino que realiza un spawn, que consiste en crear un nuevo proceso a partir de una porción del código del padre. Esta librería funciona de la siguiente manera.

Primero se crea un objeto `Process` al que se pasa una función que se convertirá en el main del proceso hijo (a diferencia del `fork()` de C en que el proceso hijo heredaba todo el código del proceso padre), así como los argumentos necesarios. En mi código la función a la que se apunta es uno de los launchers, al que se le pasa el nombre del nodo de ROS. Una vez creado el objeto, se invoca al método `.start()` para que inicie su ejecución y al método `.join()` para evitar que el proceso principal termine antes que los subprocessos hijos.

Una vez en una función launcher se configura el socket como sea necesario, se crea una instancia de la clase del listener que queremos crear, y se ejecuta el método `.run()` del objeto recién creado. A continuación se detalla el funcionamiento de cada objeto.

Funcionamiento de los objetos creados

Todos los objetos inician su ejecución en el método `.run()` definido en la clase `listenerGeneric`. Este método lanza el método `.listen()` que será distinto para cada una de las clases.

Objeto instancia de `listenerFingers(listenerGeneric)`:

Una vez entra en el método `listen()` crea un nombre para el nodo, que seguirá la convención `nombre_chat + "_listener"`. A continuación se crea el nodo en modo anónimo para que varios talkers puedan correr a la vez. Para iniciarlo se usa el método `init_node` de la librería `rospy`. En ROS los nodos tienen un nombre único que los identifica; si dos nodos tienen el mismo nombre cuando se lanzan, el más antiguo es eliminado.

Una vez creado, lo definimos como `Subscriber`, al que pasamos el nombre del chat, el tipo de dato que debe recibir, y la función de callback que va a

gestionar los datos leídos. En este primer caso, el tipo será `Float64` ya que leeremos el valor porcentual de flexión de un solo sensor.

La función de callback en este objeto lo redondea al cuarto decimal, lo serializa y lo envía a través de un socket UDP/UNIX creado por el receptor. La justificación de este tipo de sockets poco convencional es la siguiente.

Justificación uso de sockets UDP/UNIX

Son de tipo UNIX porque un tipo de endpoints de comunicación para el intercambio de datos entre procesos ejecutándose en el mismo sistema operativo de tipo POSIX, y en principio los procesos roscore y el entorno de realidad virtual está pensado que corran en la misma máquina. 7

Por ello, y sabiendo que va a ser una máquina GNU/Linux, también por especificaciones del proyecto, se ha optado por usar sockets de tipo UNIX que están pensados para la comunicación de procesos dentro de un mismo sistema UNIX, ya que la alternativa habría sido o usar otros mecanismos IPC (colas de mensajes, memoria compartida), que en caso de en un futuro quererse implementar sockets IP habrían requerido una reprogramación profunda; o usar directamente estos sockets, que funcionan más despacio ya que llegan más abajo en el stack de comunicación, y que en realidad no están pensados para comunicar dos procesos corriendo en el mismo sistema, sino que aprovecharíamos el caso particular `127.0.0.1 localhost` como se hizo en el Hito 1.

Por ello, parece mucho más razonable usar el protocolo UNIX Domain Socket ya que está específicamente diseñado para comunicar procesos dentro del mismo OS.

Además, son de tipo UDP porque cuando se trabaja en este tipo de aplicaciones de streaming, la relevancia de un dato viene dada por dos cuestiones:

1. ¿Es el dato más nuevo?: Un dato solo nos interesa si es el más actualizado. Si tenemos un dato de $t-3$, y por alguna razón se pierde el dato $t-2$ y recibimos directamente $t-1$, el dato perdido no tiene ningún interés ya, porque hay un dato más reciente, que representa mejor el estado actual del sistema.
2. ¿Hace cuánto que salió este dato?: La velocidad de transmisión es muy importante, ya que a medida que pasa el tiempo, más posibilidades hay

de que el dato no refleje la realidad fidedignamente porque haya habido algún evento que la haya alterado. Este concepto es evidente, cuando por ejemplo pasamos en una reproducción audiovisual de 30 frames por segundo (fps) a 60fps, el resultado es mucho más realista.

Para programar estos sistemas, se emplea programación en tiempo real, en la que el tiempo se divide en slots, se fijan deadlines para cada actividad, y se organizan las ejecución de forma que no se pasen estos deadlines. Existen dos aproximaciones:

- Sistemas de tiempo real “hard”: son sistemas en los que perder un deadline constituye un fallo general del sistema. Ejemplos de esto son OS como RTLinux. En nuestro caso, como Ubuntu no es un sistema de tiempo real, construir así la aplicación no es factible.
- Sistemas de tiempo real “soft”: la pérdida de deadlines influye en la calidad del resultado final. Como se puede intuir, este es nuestro caso, ya que perder datos, o momentos de sobrecarga del sistema se traducen en una peor ejecución del entorno de realidad virtual, donde se pueden percibir pequeños glitches en los que quede paralizado.

Para abordar este segundo tipo de sistemas, la filosofía que se utiliza normalmente es la de tratar de hacer todo el proceso lo más rápido posible, asegurarse de que en condiciones normales funciona de manera aceptable, evitar reenvíos y facilitar el paralelismo. Por ello se han usado sockets de tipo UDP, ya que se trata de un protocolo muy ligero, que no garantiza que los datos lleguen al sistema final, ni tiene ningún tipo de handshake. Es por eso que este protocolo se usa habitualmente en aplicaciones de comunicación por internet como VoIP (Voice over IP) y videollamadas, y es por esto que se ha considerado ideal para este Proyecto.

La instrucción `.spin()` evita que Python termine hasta que se haya parado el nodo, de forma que este sigue leyendo y transmitiendo datos hasta que recibe la señal `SIGTERM`.

Objeto instancia de `listenerExp(listenerGeneric)`:

Este objeto funciona igual que el anterior, solo que los datos recibidos no son un `Float64` sino un array de estos, de tipo `Float64MultiArray`.

El método `callback` realiza la misma operación. Serializa el array recibido con la librería `json` y lo transmite a un socket UDP/UNIX

JSON

Para solucionar el problema de la transmisión de datos, han surgido distintas librerías que implementan estándares de serialización. Uno de los más usados actualmente es JSON (“JavaScript Object Notation”). Originariamente desarrollado para JavaScript, actualmente su filosofía se ha implementado en librerías para un gran número de lenguajes de programación con mayor o menor éxito. En el caso de Python la librería de JSON es muy potente por lo que se ha decidido usar para serializar en el Hito 3.

JSON es un formato de intercambio de datos ligero, inteligible para humanos para leer y escribir, que sirve para almacenar y transmitir la información de una forma organizada y de fácil acceso. Tiene dos formas de trabajar: con una colección de pares nombre/valor, o por una lista ordenada de valores. En Python funciona de la segunda forma, que básicamente se traduce en que convierte el objeto que se le pase en un `String`, de forma que conserva la forma pero internamente está organizado en bytes. Antes de transmitir los datos se codifican con la instrucción `dumps()` y en el destino se decodifican con la instrucción `loads()` ambas pertenecientes a la librería `json` de Python.

Objeto instancia de `listenerCalibration(listenerGeneric)`:

Este objeto es muy diferente de los anteriores. Al ejecutarse el método `listen`, lo primero que hace es crear un hilo que actuará de servidor para transmitir la calibración. Para ello utiliza la librería `multithreading`, cuyo funcionamiento de cara a la configuración y el lanzamiento de los nuevos hilos es similar al de `multiprocessing` para la creación de nuevos procesos. El servidor habilita un socket TCP/UNIX y queda a la espera de nuevas conexiones para la transmisión de los datos de la calibración.

A continuación crea un nodo ROS que lee del chat de calibración máxima o del de calibración mínima. La función `callback` de este nodo abre un archivo llamado `calibration_max.txt` (o `calibration_min.txt`) en el que escribe los datos leídos, para su uso asíncrono. De esta forma nos aseguramos de que los datos de la calibración sean siempre los más nuevos, en el caso de que en el

futuro se decida añadir una función de recalibración al programa que gestiona el guante.

El servidor por su parte, cuando recibe una conexión a través del socket TCP abre el fichero que le corresponda, lee los datos de la calibración, los serializa y transmite a través del socket, y cierra la conexión. Finalmente, al tratarse de un servidor queda a la espera de nuevas conexiones. La justificación de usar sockets de tipo TCP es la necesidad que transmitir correctamente y con seguridad todos los datos de la calibración. Es importante el transmitir los datos más recientes, pero de esa gestión se encarga el método callback, manteniendo actualizado el fichero. Sin embargo, es imperioso que todos los valores de la calibración se transmitan bien. Es por ello que el protocolo TCP es el candidato ideal.

Para sincronizar los accesos al fichero se ha creado un semáforo tipo mutex como atributo de la clase `listenerCalibration`. En Python, este tipo de semáforos se llaman `Lock` y son el tipo más básico de primitiva de sincronización entre hilos. Cuando se crea un objeto tipo `Lock`, este es contenido por todos los hilos, que realizan acciones de bloqueo y desbloqueo del mismo. Un `Lock` solo tiene dos estados, bloqueado y desbloqueado (“locked” y “unlocked”), y dos métodos básicos: `.acquire()` y `.release()`.

`.acquire()` pasa al estado de bloqueo al `Lock` si el estado previo era `unlocked`, o bloquea el hilo que invoca el método en caso de que el estado sea `locked` hasta que el hilo que tenga bloqueado el `Lock` lo libere, y `.release()`, que cambia el estado de `locked` a `unlocked`.

setUpLogger:

Esta función se encarga de gestionar el logger en el que escriben todos los procesos. Es igual a la que se ha implementado en `blender_script.py` salvo por el parámetro `'name'` ya que en este caso se necesita indicar cuál es el proceso padre que lanza los subprocesos para añadirlo al nombre del archivo.

Lo primero que hace es crear el nombre del logger con la ruta `'./log/'`, el nombre del proceso padre, el nivel de escritura y la fecha.

A continuación en función del nivel entra en uno de los tres casos, y configura el archivo. El proceso padre entrará con modo `'w'` para crear un archivo nuevo, y todos los subprocesos hijos acceden con modo `'a'` para ir añadiendo al

archivo sin borrar lo que ya hay escrito.

Scripts de lanzamiento de la comunicación: main_sub.py y cal_sub.py

Para utilizar la librería `listenerGeneric.py` descrita previamente se han creado dos scripts en Python. Uno para lanzar la comunicación de los datos de normal funcionamiento del guante, `main_sub.py`, y otro para la transmisión de los datos de calibración. En ambos casos el primer paso es importar la librería creada por nosotros con la instrucción:

```
from listenerGeneric import *
```

main_sub.py

Una de las virtudes de Python es que es muy tolerante con respecto a los tipos. Uno puede crear listas de prácticamente cualquier cosa, incluso mezclando distintos tipos y tamaños. En este primer script se crea una lista vacía 'p', en la que se añadirán los objetos-proceso creados. Como se dijo antes, cada proceso es un objeto de tipo `Process` que no empieza hasta que se llama a su método `.start()`. De esta forma, se crean 10 objetos, cada uno con la cadena de caracteres correspondiente al chat al que se va a conectar el nodo subscriptor que creará en la siguiente fase, más un proceso extra que será el que gestione la comunicación de los datos de los sensores interdigitales.

Otro de los puntos fuertes de Python es que permite iterar sobre los elementos de una lista. Es decir, en vez de emplear un índice para recorrer los elementos de la lista, podemos usar los elementos en sí. De forma que con las líneas siguientes se lanzan todos los procesos:

```
for process in p:  
  
    process.start()
```

y se unen con `.join()` para, como se indicó antes, evitar que el proceso principal termine:

```
for process in p:
```

```
process.join()
```

Esta forma de lanzar los procesos será la misma siempre que se lancen procesos en este proyecto, y también cuando se lancen hilos, ya que la API de hilos y procesos es similar.

cal_sub.py

El código de este script es similar al anterior, pero en este caso solo lanzaremos dos procesos, uno para la calibración máxima y otro para la calibración mínima, ambos instancias de la clase `listenerCalibration()`.

Logger:

Finalmente, describiremos un tipo de funcionalidad que se ha añadido a todos los scripts de Python desarrollados, el logger. El logger es un archivo, normalmente gestionado a través de una librería (en este caso `logging`) en el que se escriben cadenas de caracteres con los mensajes necesarios para el estudio del funcionamiento del proceso. Es una forma más rápida, más ordenada y más profesional de debuggear un programa, pero también de hacer chequeo de errores y mantener un archivo con el historial de funcionamiento.

Consiste en:

- El archivo de logger, un archivo de texto `.log` en el que se escriben los mensajes, uno por línea, con ciertas trazas como el usuario, la fecha etc. que son configurables en tipo y formato.
- El nivel de escritura: existen cinco niveles, en orden de menor a mayor criticidad: `DEBUG`, `INFO`, `WARNING`, `ERROR` y `CRITICAL`. Cada mensaje lleva asociado un nivel de criticidad, y si este es menor al establecido para el archivo, no se escribirá.
- Los mensajes: llevan asociado un nivel de criticidad decidido por el desarrollador y una cadena de caracteres que puede contener lo que el desarrollador considere oportuno.

Este sistema es extensamente usado en la industria ya que si bien para el desarrollo resulta más lento que sencillamente ir imprimiendo por pantalla los

mensajes necesarios, a la hora de tener una aplicación funcionando resulta muy útil, ya que si ésta sufre un error o un mal funcionamiento no detectado en la fase de desarrollo, se puede acudir a estos ficheros en los que queda registrado el flujo del proceso, y se puede reenviar a los desarrolladores.

2.3 Hito 3

En este tercer hito se ha desarrollado una mano virtual en tres dimensiones en el programa Blender, así como su programación a través del Blender Game Engine (BGE).

2.3.1 Estudio preliminar de los requisitos

El objetivo es tener un modelo virtual en 3D de una mano izquierda humana genérica, que sea capaz de reproducir los movimientos de la mano real que tiene puesto el guante, a través de los datos que este transmita desde los sensores. Por lo tanto tendremos dos movimientos básicos:

- Flexión longitudinal de los dedos
- Separación o acercamiento de los dedos entre sí

Estudio anatómico de la mano y puesta en relación con el guante sensorizado

La anatomía necesaria para la elaboración del modelo articulado de la mano es la relativa a las articulaciones para las que se dispone de sensor en el guante. Dichas articulaciones son las metacarpofalángicas y las interfalángicas proximales para los dedos del segundo al quinto, y la articulación carpometacarpiana del pulgar, metacarpofalángica e interfalángica del primer dedo.

Dado que no disponemos de sensor para medir la flexión de las articulaciones interfalángicas distales, en el modelo articulado de la mano heredan su movimiento de las interfalángicas proximales puesto que su movimiento independiente es mínimo.

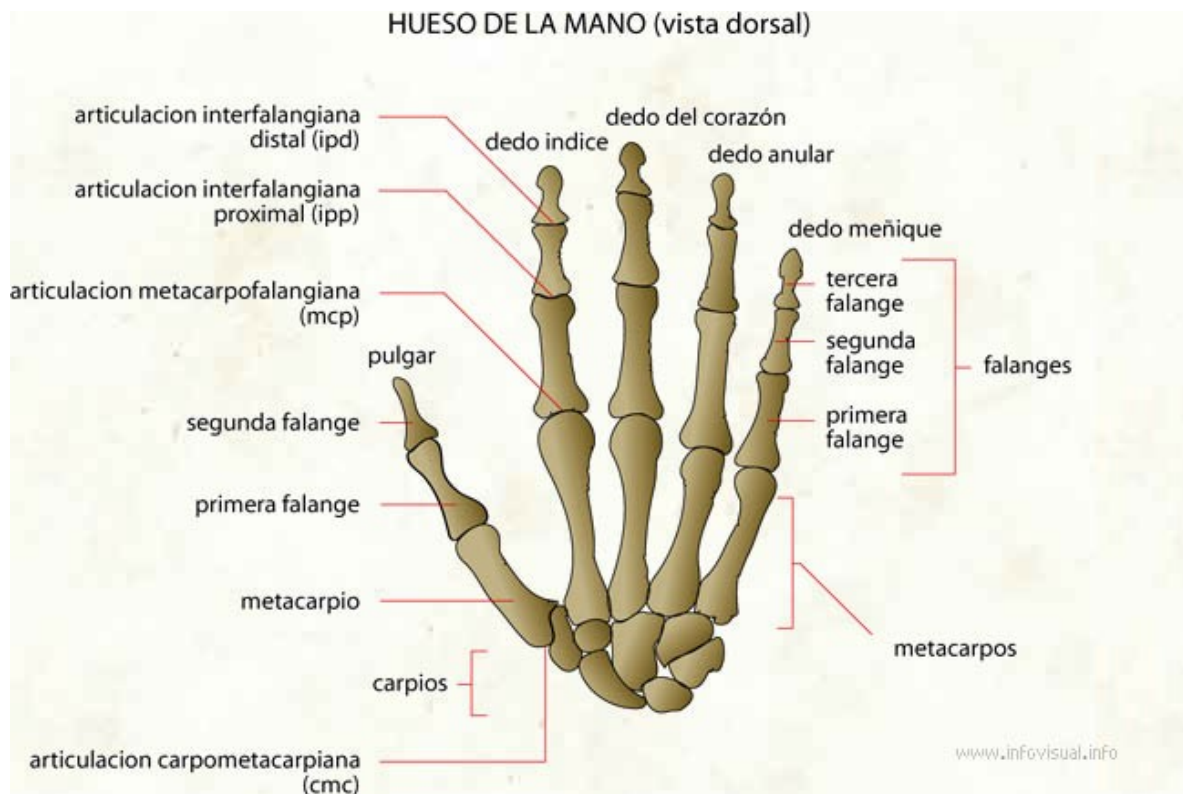


Figura 19: Huesos y articulaciones de la mano derecha, vista dorsal

2.3.2 Modelo 3D en Blender

Como se dijo en el apartado 2.2- Modificaciones – Blender, este integra toda la toolchain para el desarrollo de aplicaciones en 3D. Para este proyecto se van a usar gran parte de las funcionalidades que proporciona, fundamentalmente en el campo del desarrollo, haciendo poco hincapié en el diseño.

Etapas del desarrollo:

- I. Modelo de la mano 3D
- II. Creación de los huesos y articulado
- III. Campo de acción y setup de las propiedades de la malla y la armadura
- IV. Desarrollo del script
 - A. Setup de las propiedades de la escena y configuración del script
 - B. Comunicación con el exterior

C. Hilos

D. Interacción con la escena

E. Auxiliares

i. LUT

ii. Calibración

iii. Logger

Modelo de la mano en 3D

Para empezar esta parte del proyecto se necesitó un modelo de la mano izquierda en 3D. Originalmente el modelo que se iba a usar era el de shadow hand, que como se vio en el apartado 2.1.1- Materiales del planteamiento inicial - Gazebo y Shadowhand, quedó descartado por razones técnicas, por lo que surgió la necesidad de crear o adquirir uno.

En Blender, los modelos son construcciones de triángulos dispuestos en tres dimensiones que crean una superficie abierta o cerrada. Quedan definidos por tanto por las coordenadas de los vértices que los forman y las relaciones entre ellos. Esto es lo que se denomina malla o 'mesh'.

Tras estudiar el diseño en Blender, se concluyó que era más útil obtener un modelo virtual libre, es decir con una licencia de libre uso y libre distribución, realizado por un artista experto en Blender, ya que el objeto de este Proyecto es el control del interfaz, y no el aprendizaje de un entorno de diseño.

Como ya se mencionó, una de las grandes virtudes del código abierto es la facilidad que hay para crear y compartir contenido con la comunidad en internet, por ello tras una investigación minuciosa se llegó a la decisión de usar el modelo Free Realistic Hand publicado en la página blendswap.com por el usuario SuperDasil. Esta mano tiene varias características que la hacen especialmente adecuada:

- 1- El modelo de la mano es indudablemente bueno y realista
- 2- Sin embargo, se trata de un “mesh” vacío. Es decir, es una malla o modelo 3D deformable pero sin ningún tipo de añadido, ni huesos, ni

restricciones, ni propiedades. Por ello, ha sido perfecta para empezar a construir desde el principio, y solo eliminar la etapa más puramente artística del diseño.

3- Está licenciada con una licencia CC-BY. Esta licencia se basa en los siguientes puntos:

- a. **Compartir:** cualquiera puede copiar y redistribuir el material en cualquier formato y medio.
- b. **Adaptar:** el material puede ser mezclado, transformado y construido como se quiera con lo que se quiera para cualquier propósito, incluido el comercial.
- c. **Atribución:** Se debe dar el crédito adecuado al autor, proporcionar un link a la licencia e indicar los cambios que se han hecho, pero sin sugerir que el licenciante apoya al licenciado ni el uso que éste le haya dado al material.
- d. **No añadir restricciones:** En ningún caso se pueden aplicar medidas legales o tecnológicas que restrinjan a otros de hacer cualquier cosa contemplada por esta licencia.

El modelo se proporciona en diferentes formatos de archivos:

- .3ds → formato de COLLADA (Collaborative Design Activity). En general la calidad de los bordes es baja, lo que perjudica la visibilidad de los movimientos.
- .dae → formato autodesk 3ds Max. Muy anguloso, da una sensación de baja resolución e irrealidad.
- .fbx → superficie muy suave. Filmbox, formato propietario de Autodesk.
- .x3d → tercera generación de Virtual Reality Modeling Language, que ofrece compatibilidad retroactiva total. Respecto a calidad es totalmente adecuada, por lo que se ha decidido usar este archivo.

Una vez decidido el modelo, se adaptó a las necesidades del proyecto. La más inmediata comprobar que efectivamente se podía crear una simulación de la mano. Para ello, era necesario encontrar la forma de deformar el mesh.

2.3.3 Creación de la armadura

En Blender una armadura es un tipo de objeto usado para articular una malla, es el equivalente a un esqueleto real y como tal está compuesto de uno o más huesos. Estos huesos se relacionan entre ellos por enlaces de fijación física o a distancia con un offset, y se pueden mover relativamente a estos enlaces igual que una articulación real. En Blender, los huesos disponen de los mismos atributos que cualquier otro objeto gráfico, por lo que es sencillo trabajar con ellos:

1. Un centro, y una posición dada por tres coordenadas y un factor de escala que forman un cuaternio.
2. Tiene un bloque de datos "Object Data" que puede ser editado en Edit Mode.
3. Soporta links a otras escenas por lo que puede ser reutilizado en múltiples objetos, aunque no es nuestro caso.
4. Sus propiedades no se ven afectadas por el uso que se le de, ni en Pose Mode ni, en nuestro caso, en el Blender Game Engine. Esto significa que la posición inicial, las restricciones etc. que se le den en el modo edición (Edit Mode) no se verán afectadas por su uso en el BGE.

Huesos

Los huesos son la unidad básica de una armadura. Constan de las siguientes partes:

- La unión inicial llamada raíz o cabeza.
- El cuerpo
- La unión final llamada cola

La posición de un hueso está definida por las posiciones de la cabeza y la cola respecto del origen. Cada hueso se mueve relativa o absolutamente, causando que la malla que lo rodea se deforme en función del área de influencia del hueso. A continuación se detalla el proceso seguido para construir el esqueleto de la mano.

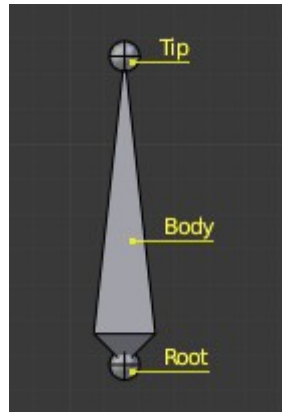


Figura 20: Partes del hueso en Blender

Creación de la armadura

Lo primero es importar el modelo de la mano, centrarlo y configurarlo como origen. Para ello se usa 'Shift+C' para centrar el cursor, 'Object → Transform → Origin to Geometry', lo que mueve el punto de pivotaje a la mano y pulsamos botón derecho 'Selection to Cursor'. A continuación con 'Alt+R' y pulsando la letra del eje sobre el que se desea rotar, colocamos el modelo de forma que la base de la mano esté orientada en el sentido Z, y el plano X-Y sirva de base. Esta orientación se ha escogido solo porque resulta más intuitivo trabajar con este objeto con el plano XY de base, y el eje Z representando la altura, pero lo cierto es que no influye demasiado.

En la imagen se puede ver la mano orientada. Además, el rectángulo negro corresponde a la cámara y el círculo negro a la iluminación de la escena. Estos elementos se han configurado para proporcionar una correcta visualización de la mano desde el frente.

El siguiente paso es iniciar la creación de huesos. Para ello vamos a 'Add → Armature → Single Bone' lo que genera un hueso en el origen. Para modificar su forma y tamaño pulsamos click derecho sobre el hueso y pasamos a 'Edit Mode' en el menú inferior. Con el botón derecho seleccionamos la cola del hueso y la rotamos para que aparezca alineado con la dirección de la mano. Este es el hueso base, que está fijo y respecto al cual se articula el resto de los huesos.

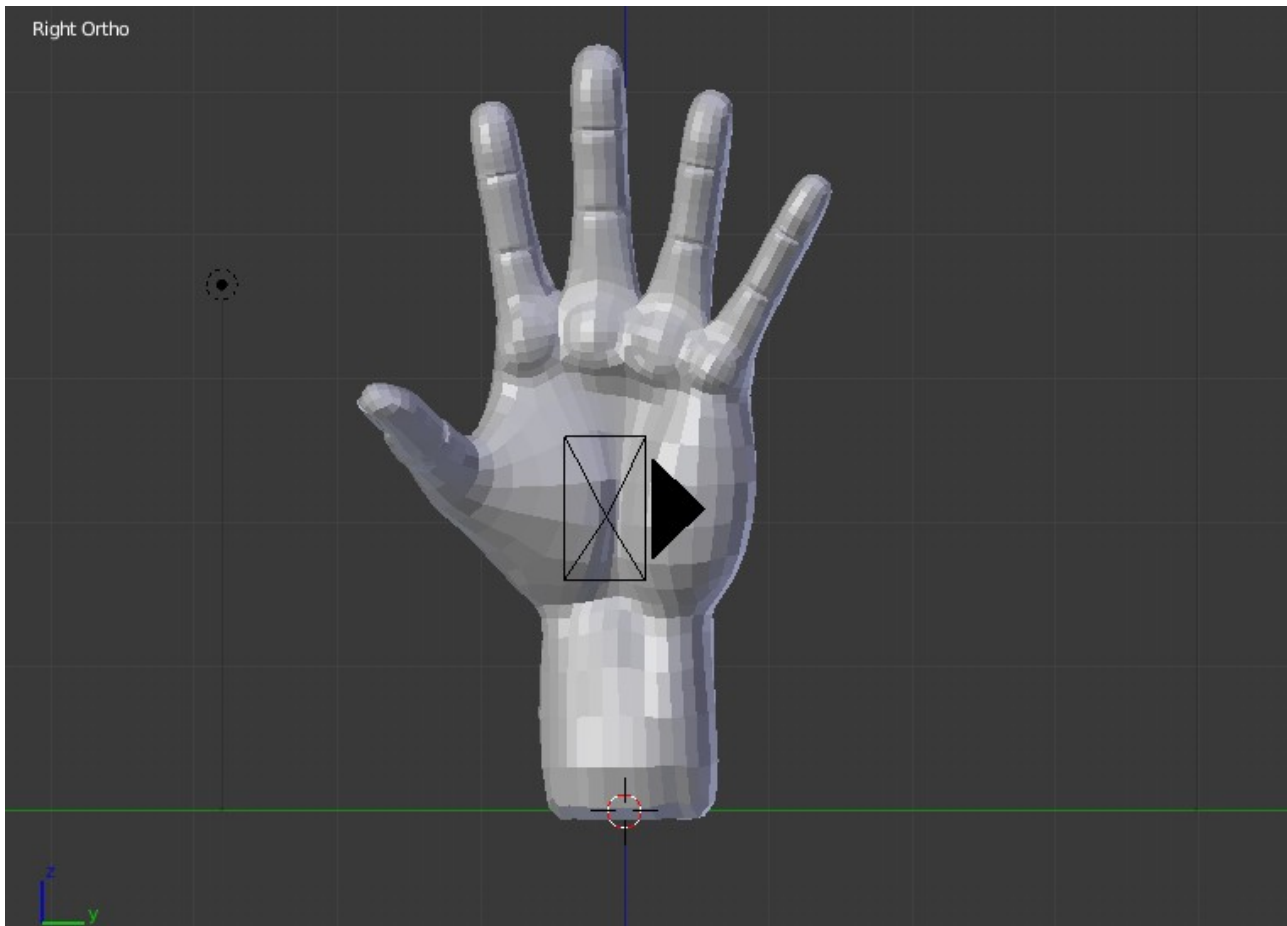


Figura 21: Modelo de la mano, con la cámara y el punto de luz fijados

En el panel de la derecha, seleccionando en el icono con un dibujo de un hueso, podemos editar ciertas propiedades de este. A todos los huesos les daremos un nombre que siga el patrón que se ha considerado más adecuado que es [sección.posición] donde sección es la parte de la mano en que se encuentra y posición la posición que ocupa dentro de la misma. En este primer hueso, la sección es toda la mano, y la posición es la muñeca, por lo que el nombre es 'hand.wirst'.

Una vez creado este hueso, lo duplicamos para crear otro hueso seleccionando el 'hand.wirst' y pulsando 'Shift+D'. Lo colocamos aproximadamente en el la posición que deseemos y lo rotamos para que quede dentro del mesh y en su posición final. Para comprobar que no se sale de la mano, activamos y desactivamos el botón 'x-ray' en el panel de la derecha, en el icono de la armadura. Este hueso, y el resto que se crearán por este mismo sistema se nombrarán siguiendo el patrón previamente definido, donde sección es el dedo al que pertenecen, y posición el hueso al que sustituyen. Por ejemplo, el hueso de la falange proximal del corazón se llamará 'middle.p_phalanx'

Para crear todos los huesos de una forma más rápida, duplicamos el hueso de la muñeca y colocamos el nuevo hueso a lo largo de cada dedo, de forma que el dedo tenga un solo hueso. A continuación pulsamos 'w' y seleccionamos 'Subdivide', el hueso queda dividido en dos, pero con una relación de parentesco entre sí, y unidos por una articulación en el punto en que se han dividido.

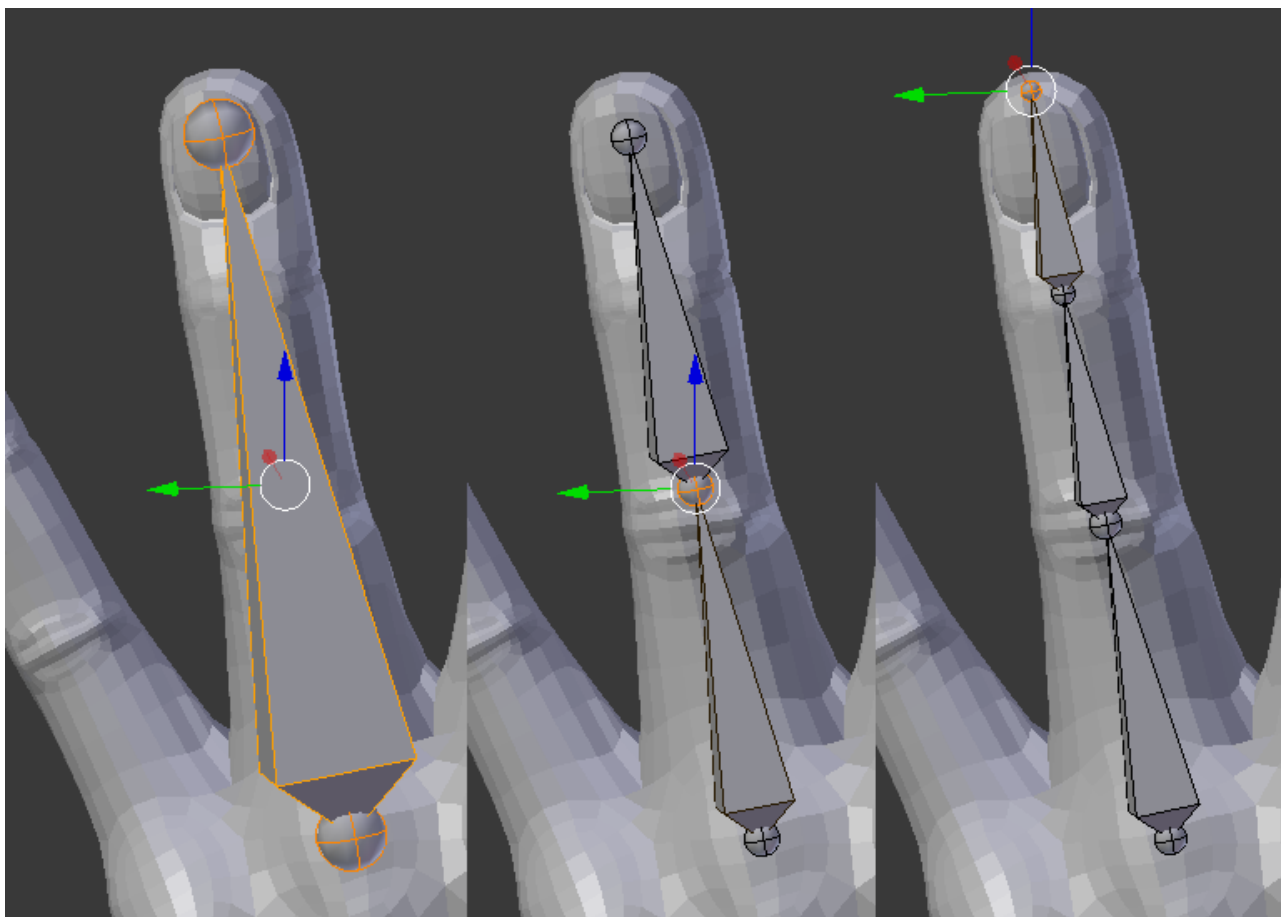


Figura 22: Proceso de división del hueso de un dedo

Estas articulaciones y relaciones de parentesco son lo que realmente nos va a permitir simular el movimiento de la mano con una alta fidelidad.

En la imagen se muestran las relaciones de parentesco entre los huesos. Los que tienen un recuadro rojo son aquellos que además tienen su movimiento supeditado al del padre.

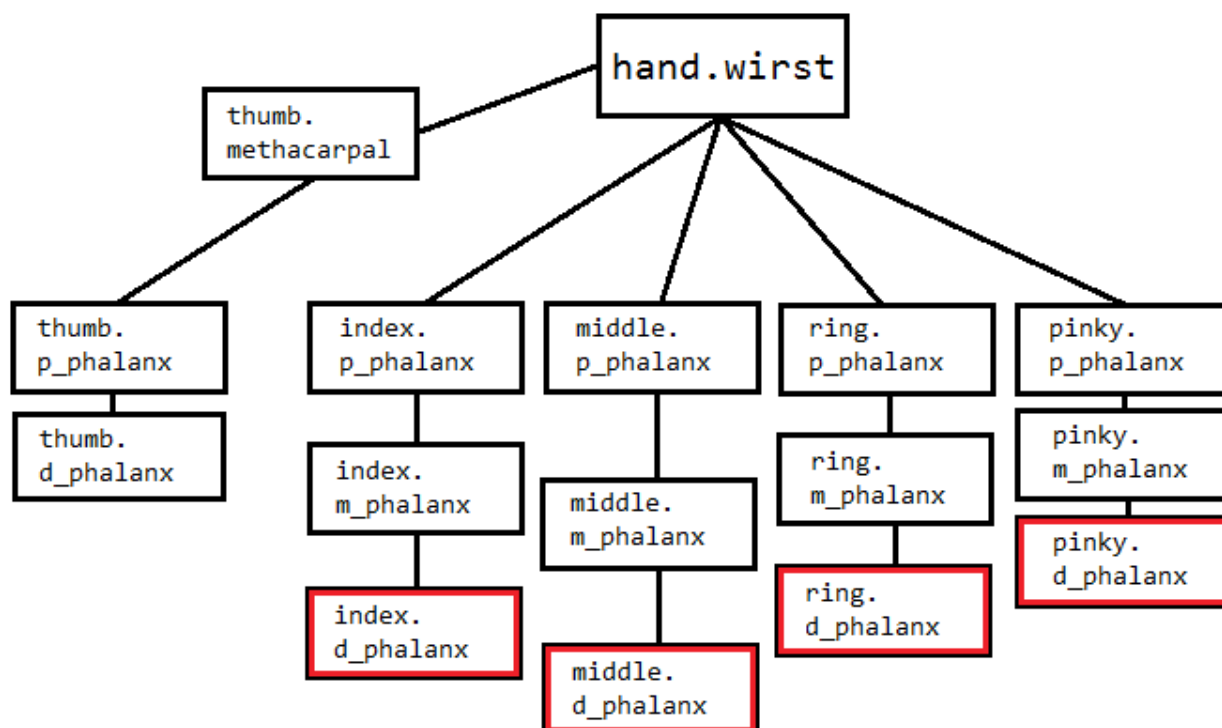


Figura 23: Esquema de parentescos y dependencias

Este proceso se realiza para todos los dedos y se ajustan las articulaciones para que coincidan con las articulaciones del modelo. Esto nos deja cinco dedos independientes entre sí, y un hueso de muñeca sin relaciones. Para relacionarlos todos lo que haremos será crear relaciones de parentesco. Para ello pulsamos botón derecho sobre el que será hijo, manteniendo 'Shift' botón derecho sobre el futuro padre y pulsamos 'Ctrl+P'. En el menú que nos aparece se nos dan dos opciones, unirlos o mantener la distancia. En nuestro caso, solo están unidos los huesos de los dedos entre sí, pero respecto a la muñeca todos guardan un offset.

Finalmente, hacemos un ajuste fino de los huesos para colocar sus ejes de coordenadas relativos en la orientación correcta. Cada hueso dispone de un eje de coordenadas local en su cola, pero la forma de rotar es respecto a ese eje de coordenadas trasladado a la cabeza del hueso.

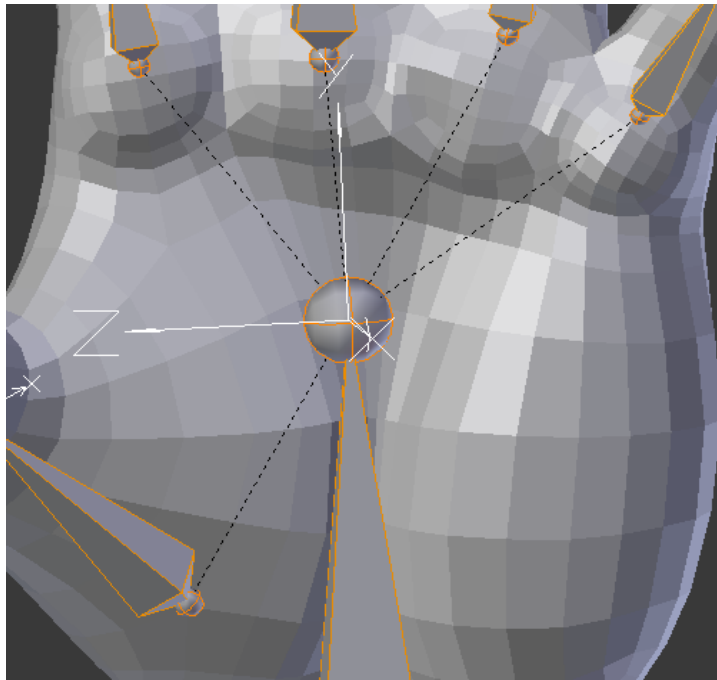


Figura 24: Detalle relación de parentesco con offset

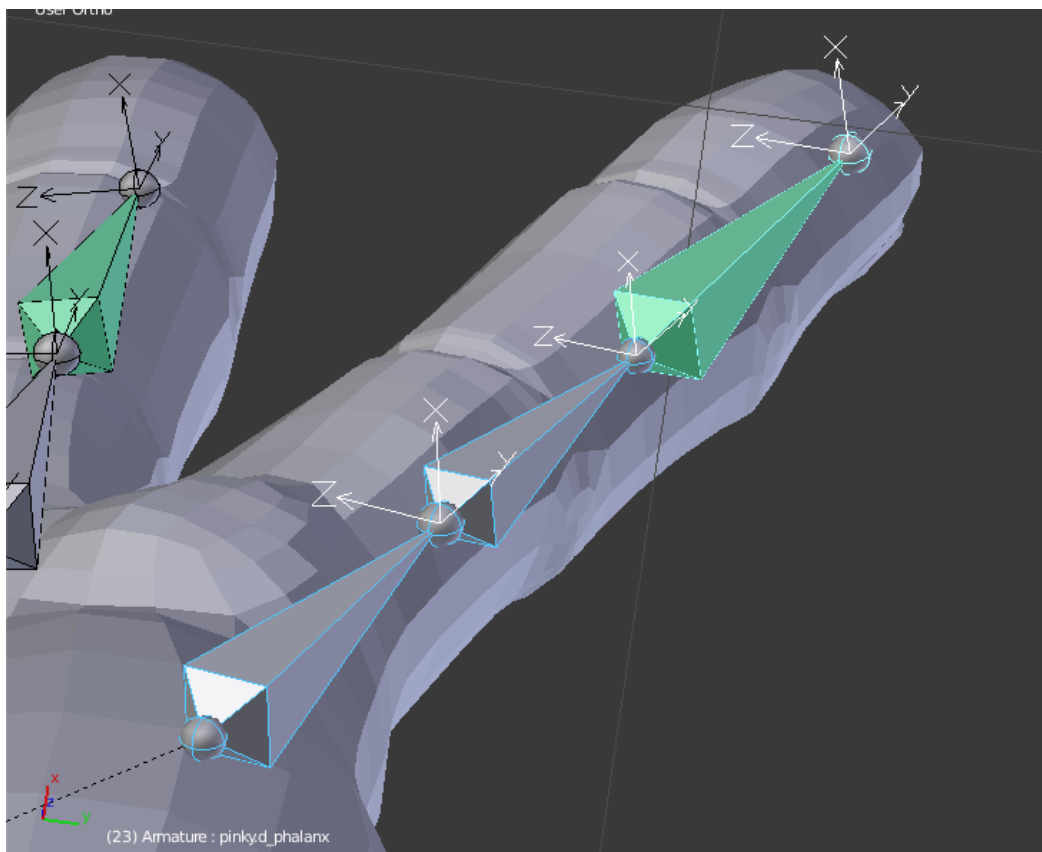


Figura 25: Detalle ejes locales de los huesos de dos dedos

Para que la armadura deforme a la malla de una forma lógica, es decir que cada hueso tenga el mismo área de influencia sobre el modelo que el hueso real tiene sobre la mano, pasamos a 'Object Mode' seleccionamos la malla, manteniendo 'Shift' seleccionamos la armadura, pulsamos 'Ctrl+P' de parentesco y elegimos 'With automatic weights'. Esto da a cada hueso un área de influencia sobre la malla, con un gradiente de control de forma que cuanto más se aleje del hueso menos influencia tenga. Al grupo de triángulos que quedan influenciados por el hueso se lo llama 'Vertex Group' y el peso con el que son influenciados queda reflejado en lo que se llama 'Weight paint'.

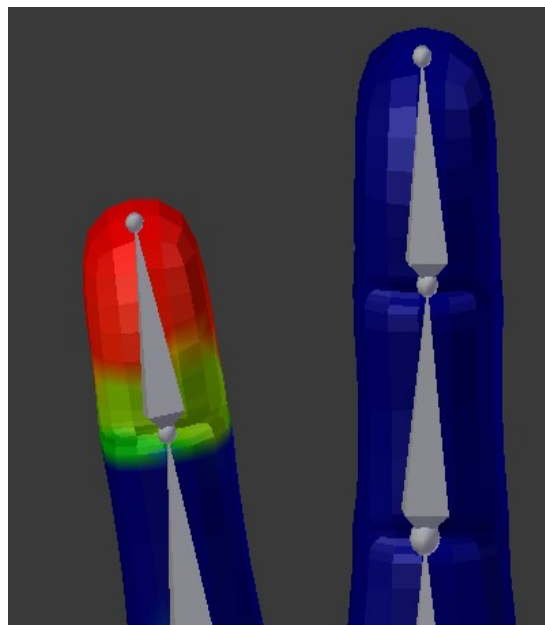


Figura 26: Detalle "Weight paint" en la falange distal del dedo índice

Si somos afortunados no tendríamos que hacer ningún cambio sobre lo que se ha generado automáticamente. Sin embargo, en este caso se han tenido que hacer variaciones sobre los pesos, para darle un resultado más realista a los movimientos de la mano.

A continuación se ajustan las propiedades de los huesos. Cada hueso tendrá ciertos grados de libertad, y tendrá otros restringidos, en función de su posición. A continuación una tabla con los huesos y sus grados de libertad.

Nombre Hueso	W	X	Y	Z
hand.wirst	X	X	X	X
thumb.methacarpal	X	V	X	X

Nombre Hueso	W	X	Y	Z
thumb.p_phalanx	X	X	X	V
thumb.d_phalanx	X	X	X	V
index.p_phalanx	X	V	X	V
index.m_phalanx	X	X	X	V
index.d_phalanx	X	X	X	V
middle.p_phalanx	X	V	X	V
middle.m_phalanx	X	X	X	V
middle.d_phalanx	X	X	X	V
ring.p_phalanx	X	V	X	V
ring.m_phalanx	X	X	X	V
ring.d_phalanx	X	X	X	V
pinky.p_phalanx	X	V	X	V
pinky.m_phalanx	X	X	X	V
pinky.d_phalanx	X	X	X	V

Figura 27: Tabla GDL de cada hueso

Finalmente, debemos tener en cuenta cómo es nuestro hueso y cómo son los movimientos de la mano humana. Si bien tenemos tres falanges por dedo (salvo en el pulgar) y estas articulaciones deben quedar reflejadas en el modelo, también es verdad que solo son independientes las falanges proximal y media, siendo el movimiento independiente de la falange distal casi nulo, ya que queda supeditado al movimiento de la falange media casi totalmente. Los creadores del guante sensorizado entendían esto por lo que colocaron solo dos sensores por dedo, por lo que solo tendremos datos de la flexión de las dos primeras falanges. Para representar en Blender el movimiento de la tercera, se ha creado una relación de dependencia con el movimiento de la segunda de la siguiente manera.

En 'Pose Mode' seleccionamos el hueso que va a ser maestro, manteniendo 'Shift' seleccionamos el esclavo y pulsamos 'Ctrl+Shift+C' y seleccionamos 'Copy rotation'. Esto añade una restricción al hueso esclavo, que ahora se moverá en función del hueso maestro. Estas restricciones se pueden ver y configurar manualmente en el menú de la derecha, en el icono del hueso y la cadena. En este caso se debe configurar el espacio a 'Local Space' ya que cada hueso se mueve respecto a unos ejes relativos, tras hacer pruebas de realismo se ha decidido que la influencia ideal es de un 90%.

Con esto quedaría la mano configurada en el plano físico y lista para empezar su programación.

2.3.4 Programación del interfaz

Una vez desarrollada la parte gráfica el objetivo es programarla adecuadamente para que reciba los datos del guante leídos de ROS por el script `main_sub.py` creado en el Hito 2. Ya en la introducción 2.2- Modificaciones - Blender se indicó que este dispone de una API de Python, su propio núcleo de Python 3.5, y una interfaz gráfica para facilitar la programación. A continuación se detalla el procedimiento seguido y se explica el funcionamiento del script que ha resultado.

Visión general de la programación

El objetivo es crear un script que a través de unos sockets UDP/UNIX lea los datos transmitidos desde ROS por el script `main_sub.py` y los utilice para colocar la mano virtual en la misma pose que tiene la mano real, con la menor latencia posible, pero también con la mayor precisión.

Para ello se utiliza el motor de videojuegos de videojuegos Blender Game Engine. Como se expuso 2.2- Modificaciones - Blender este es una herramienta para proyectos de tiempo real soft, desde visualizaciones arquitectónicas hasta simulaciones y juegos. Con él se pueden crear aplicaciones o simulaciones 3D interactivas, es decir BGE va renderizando las imágenes continuamente en tiempo real, e incorpora facilidades para la interacción con la escena durante el proceso de renderizado, de las cuales hablaremos a continuación.

El editor lógico

Por defecto, desde el propio interfaz de Blender el usuario puede acceder a un editor lógico de muy alto nivel, conducido por eventos, que permite controlar la escena por medio del trío de bloques lógicos 'Sensor' + 'Controller' + 'Actuator'.

Como este es un campo muy amplio, en el presente documento se ceñirá la explicación a los elementos utilizados. El lector interesado puede ver en la

documentación todos los bloques lógicos disponibles.

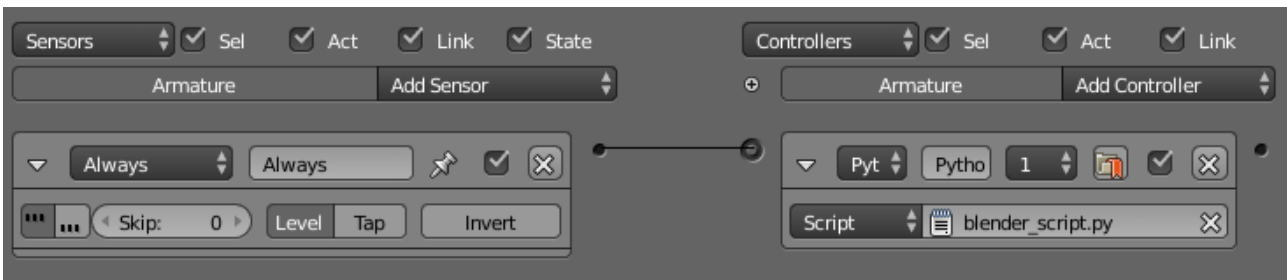


Figura 28: Detalle editor lógico de Blender

El control de la escena es, como se ha indicado, conducido por eventos. Por ello, se ha utilizado como sensor un bloque lógico tipo 'Always' que genera una señal de reloj cuadrada clásica 50/50. Esta señal de reloj dispara en cada flanco de subida un bloque lógico controlador 'Python'. Este bloque está configurado sobre la armadura creada en el apartado anterior. Con él, podemos enlazar un script de Python 3.5 a la escena, extraer las propiedades de la misma, usarlas y modificarlas, y escribir los resultados de los cálculos sobre la misma, de forma que en el siguiente ciclo de renderizado se apliquen los cambios efectuados. Debido a esta propiedad del script de actuar sobre la escena, no es necesario un bloque lógico de tipo actuador, ya que el controlador hará esa función también.

El script creado, al que se ha llamado `blender_script.py`, está en la misma ubicación que el resto de scripts. Si bien Blender incluye una herramienta de edición de código, no es realmente muy potente, por lo que permite enlazar scripts de fuera del entorno, para poder editarlos externamente e importar los cambios fácilmente a través de la combinación 'Alt+R'.

Por lo tanto, una vez en situación de empezar a programar, el esquema lógico de bloques que queda es el siguiente:

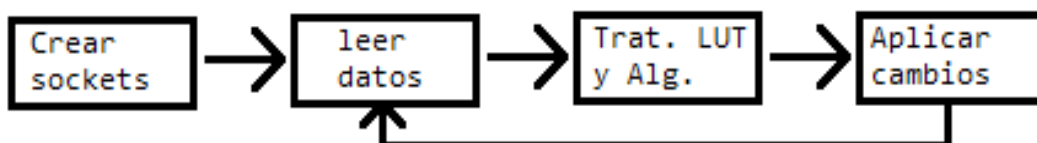


Figura 29: Proceso realizado por los hilos

Como último dato antes de pasar a explicar el script, hay que exponer cómo funciona una escena de Blender. BGE funciona como un gran objeto, que comprende todos los demás objetos que forman parte de la ejecución, y que dispone de los métodos adecuados para acceder a ellos. De estos, interesa el controlador general de la escena, para poder manipular esta. Para acceder a ella, importamos el controlador al script, y accedemos al 'owner' de la escena con las siguientes dos líneas de código:

```
cont = bge.logic.getCurrentController()  
  
hand = cont.owner
```

'hand' contiene las propiedades de la escena que se está procesando, que van desde los huesos hasta propiedades que configure a mano el programador. A continuación se explica el desarrollo del script, y cómo esto es importante, tanto para los sockets como para los movimientos de la mano.

Sockets

Al tratarse de sockets UDP, éstos deben ser creados por la entidad que lee de ellos, en nuestro caso el script. Como se indicó previamente, el script se ejecuta una vez por ciclo de reloj, por lo que se estarían creando y destruyendo continuamente.

Aquí es donde entra en juego el 'owner' de la escena explicado en el apartado anterior. Podemos crear una propiedad del juego en forma booleana que sea falso al iniciarse, pero que en la primera ejecución pase a ser verdadera, de forma que si es falsa configure todos los sockets y el resto de procedimientos que se explicarán en el apartado Funciones Auxiliares más adelante dentro de este mismo epígrafe, que solo deben ser ejecutados una vez al inicio, y las siguientes veces pase a ejecutar el funcionamiento normal.

Esta propiedad se crea en la interfaz gráfica de Blender, en el entorno Blender Game. En este caso se ha creado una variable booleana de nombre 'startUp' que tenga el valor 'False' al inicio.

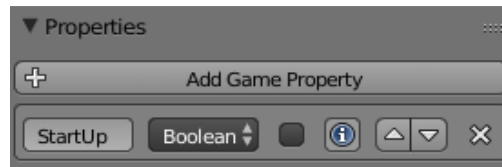


Figura 30: Detalle editor de propiedades del juego de Blender

Para acceder a esta propiedad, al ser una propiedad del juego, simplemente la buscamos por diccionario en hand, y evaluamos su nivel lógico:

```
if not hand['StartUp']:  
    #Startup code  
  
    hand['StartUp'] = True  
  
else:  
  
    #non-startup code
```

Esto soluciona el primero de los problemas, ya que solo creamos los sockets en la primera ejecución. Sin embargo, no solo necesitamos esto sino que necesitamos cargarlos en persistencia de alguna manera, de forma que pasen a formar parte del juego globalmente, no solo de la escena, y por ello podamos acceder a ellos en las escenas siguientes.

Para hacer esto, podemos añadirlos a la librería 'Game Logic' de forma temporal en nuestra ejecución. Esta librería es la que globalmente contiene una colección de los componentes lógicos de nuestra simulación que controlan su comportamiento. Por ello, es persistente a lo largo de toda la simulación. Además, es necesaria en el script para ser usada por otras funciones, así como por algunas constantes que se explicarán más adelante en el apartado Funciones Auxiliares más adelante dentro de este mismo epígrafe.

Para usarlo, simplemente lo importamos y añadimos los elementos manualmente a través del operador '.'.

```
import GameLogic as GL  
  
GL.nombre_para_GL = elemento_a_añadir
```

Por lo tanto, la solución final adoptada para los sockets es la siguiente:

1. Primero se crea una lista con los objetos tipo socket necesarios.

```
s = [socket.socket(socket.AF_UNIX, socket.SOCK_DGRAM) for
      i in range(num_sockets)]
```

2. A partir de una lista que contiene los nombres que se usarán para los sockets (ver apartado 3.2.2- Relación y descripción de los ficheros generados - Justificación uso de sockets UDP/UNIX) se crea una lista de direcciones de los sockets, y se comprueba que no estén ya creados en la dirección '/tmp', y en caso de estarlo, se eliminan con el método `os.unlink()`.

```
os.unlink(s_address)
```

3. Finalmente se enlazan todos los sockets de la primera lista con las direcciones de la segunda con el método `.bind()` y se configuran para que no sean bloqueantes. Esto es porque como se indicó en 3.2.1 Justificación del interfaz ROS-Entorno virtual, Blender no es multiproceso, de manera que si se bloquea el script para esperar la llegada de datos, se bloquea efectivamente todo el proceso 'blender'.

```
for i in range(num_sockets):
    s[i].setblocking(0)
    s[i].bind(socket_address[i])
```

4. El último paso es añadir los sockets a GL:

```
GL.s = s
```

Una vez creados los sockets, pasamos a utilizarlos en la siguiente fase fundamental, la lectura de la información que transmiten.

2.3.5 Hilos de procesamiento

Para hacer la lectura de los datos y su aplicación sobre la escena, se ha optado por separar el procesamiento de cada hueso en un hilo. Las razones para hacer esta separación en hilos son las siguientes:

1. Conceptualmente, es más sencillo el tratamiento de cada hueso independientemente, ya que cada uno tiene su propio contexto:
 - a) Un socket.

- b) Un dato para `moverZ`, un array para `moverX`.
 - c) Un hueso asociado para `moverZ`, un array para `moverX`.
2. Si un hilo falla, bien en la lectura o en la escritura por la razón que sea, es más sencillo acotar el problema, no afecta al resto de hilos en proceso, y permite un debuggeo más rápido.
 3. Las ventajas propias de un proceso multihilo para este caso.
 - a) Permite a la aplicación permanecer respondedora pese a posibles bloqueos.
 - b) Ejecución más rápida en sistemas multiprocesador, debido al paralelismo resultante.

Cada hilo ejecuta el esquema descrito previamente en esta sección. Recibe un socket, del cual leerá un dato, que tratará adecuadamente para escribir un valor válido en el hueso, y escribirá este valor de rotación.

Proceso de lanzamiento

Existen dos clases de hilos:

1. Hilos para el movimiento de los dedos respecto de su eje Z. Lee un dato y lo escribe sobre un hueso. Se crean diez y su target es la función `moverZ`.
2. Un hilo para el movimiento de expansión de la mano, que actúa sobre los huesos moviéndolos alrededor de su eje X. Su target es la función `moverX`.

Para lanzarlos se crean en un bucle y se almacenan en una lista los `mover_threads`.

Una vez almacenados, se lanzan por medio de la instrucción `.start()` y se sincronizan por medio de la instrucción `.join()`. De esta forma el hilo principal y por lo tanto Blender esperará a que todos hayan terminado antes de cerrarlo.

A continuación se desarrolla el proceso que lleva a cabo un hilo independiente.

Hilo tipo moverZ

Para lanzar un hilo de este tipo se deben pasar una serie de argumentos:

- `thread_bone`: hueso en el que va a escribir los datos leídos.
- `owner`: el 'owner' de la escena, que antes se ha almacenado en la variable 'hand'. Se pasa para poder modificar la escena.
- `socket`: socket del que se va a leer. Se pasa el socket completo, extraído en el momento del lanzamiento del hilo de la lista `GL.s`.

Una vez lanzado, este hilo lee con la instrucción `.recvfrom()` del socket y en caso de haber recibido datos se pasan a la función `applyRotationZ()` donde se escriben los datos sobre los huesos. Finalmente se actualiza la escena con la instrucción `.update()` lo que hace efectivos los cambios realizados dentro de la función `applyRotationZ()`.

Función `applyRotationZ(bone, data)`:

Dentro de esta función se mueve el hueso el ángulo necesario leído del socket. Recibe dos argumentos:

- `bone`: hueso a mover.
- `data`: valor de la rotación en porcentaje.

Al recibir el valor de la rotación en porcentaje utiliza la Look Up Table número 1 creada en la fase de primera ejecución para acceder a su valor en radianes, ahorrando así tiempo de cálculo.

Configura el sistema de rotación como rotación respecto a los ejes XYZ aplica el valor al hueso.

Hilo tipo moverX:

A diferencia de la tipología de hilos estudiados en el apartado anterior, solo se creará un hilo que gestionará a la vez la movilidad de cinco huesos simultáneamente.

Los parámetros que se le pasan son:

- `owner`: el `'owner'` de la escena, que antes se ha almacenado en la variable `'hand'`. Se pasa para poder modificar la escena.
- `socket`: socket del que se va a leer. Se pasa el socket completo, extraído en el momento del lanzamiento del hilo de la lista `GL.s`.

Los huesos sobre los que actúa están almacenados en el array `'x_bones'` que se crea en una función auxiliar al inicio de la ejecución (ver Funciones Auxiliares más adelante en este mismo epígrafe).

El funcionamiento es similar a `moverZ`, solo que en este caso la información leída es un vector que contiene los valores de los ángulos de los cuatro sensores interdigitales. Debido a diferencias de serialización y procesamiento de la información entre Python 2.6 (usado en `listenerGeneric.py`) y Python 3.5 (usado en Blender) la información se envía serializada con JSON, pero al recibirse debe codificarse de nuevo como String UTF-8 ya que los sockets de 2.6 toleran el envío en caracteres pero para 3.5 todo lo que lee son bytes.

Una vez hecho esto se llama a la función `applyRotationZ`.

Función `applyRotationZ(data)`:

Esta función recibe el string serializado con JSON con los datos de los sensores y lo deserializa. A continuación establece como modo de rotación el euleriano en todos los huesos e inicia un proceso de comprensión de la información transmitida a través de un algoritmo de posicionamiento propio.

Algoritmo de posicionamiento:

El posicionamiento de la mano en el espacio es un tema que puede dar lugar a trabajos enteros, por lo que se ha desarrollado un algoritmo sencillo para, a partir de los datos de los sensores interdigitales, poder obtener la pose aproximada de la mano.

El algoritmo se basa en el uso de los valores relativos para averiguar la orientación de los dedos, y a partir de ahí aplicar los ángulos en la dirección correcta. Los pasos son los siguientes:

1. El dedo pulgar es independiente del resto de la mano, por lo que se puede aplicar el valor directamente sobre él.
2. Se restan los valores en radianes del sensor Índice_corazón del sensor Corazón_anular, y lo que resulta es una primera rotación.

$$\text{Índice_corazón} - \text{corazón_anular} = R1$$

3. Si este movimiento sale positivo, significa que el dedo corazón está más inclinado hacia el anular, y viceversa.

1. R1 queda asignado a los dedos corazón, anular y meñique.
2. Restamos al valor del sensor Índice_Corazón el valor de R1, ya que es un valor que ya se está aplicando al dedo corazón, y el resultado se aplica sobre el dedo Índice.

$$\text{Índice_corazón} - R1 = R2$$

3. Se aplica el valor del sensor Corazón_anular sobre los dedos anular y meñique, ya que se verán desplazados por la posición del dedo corazón
 4. Se aplica el valor del ángulo del sensor Anular_meñique sobre el dedo meñique.
4. Si R1 sale negativo, significa que el corazón está más inclinado hacia el dedo índice:

1. Se aplica el valor de R1 sobre el corazón y el índice.
2. Se resta al valor del sensor Corazón_anular el valor de R1, porque esta rotación ya se ha aplicado, y se aplica la diferencia al anular y al meñique.
3. Se aplica el valor del sensor Índice_corazón sobre el dedo índice.
4. Se aplica el valor del ángulo del sensor Anular_meñique sobre el dedo meñique.

Valores Recibidos = {P1, P2, P3, P4}
 Rotaciones = {R1, R2, R3, R4, R5}
 Dedos = {D1, D2, D3, D4, D5}

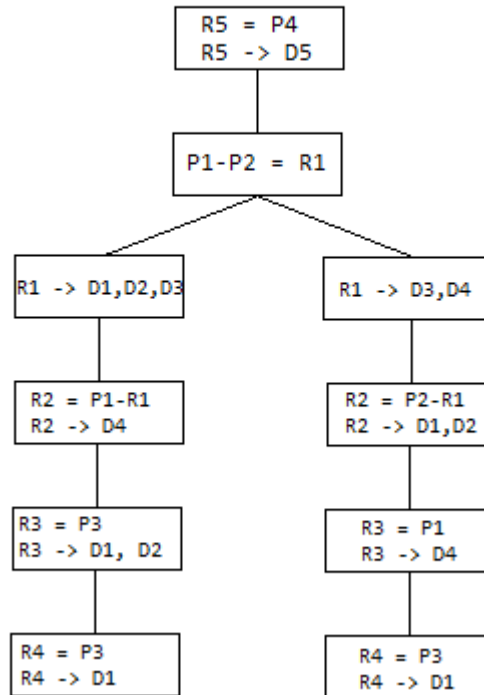


Figura 31: Algoritmo de posicionamiento

Este algoritmo no busca representar fielmente los movimientos de la mano. Originalmente se iba a aplicar un sistema todo-nada en el que si un sensor valía más del 50% esos dedos estuvieran cerrados, y viceversa. Sin embargo, para dar algo más de realismo se decidió aplicar este algoritmo de desarrollo propio.

Funciones auxiliares:

En este último apartado se exponen las funciones auxiliares presentes en `blender_script.py` que se han realizado para acelerar y modularizar su funcionamiento.

setupLoggerFile(level):

Esta función es similar a la creada para la librería `listenerGeneric.py` (ver apartado 3.2.2 Relación y descripción de los ficheros generados)

- Biblioteca `listenerGeneric.py`). Su función es la de gestionar el fichero de logger. Al igual que para los procesos independientes, cada hilo debe abrir el archivo independientemente. También crea otra lista con los huesos que se moverán en de forma expansiva.

```
GL.bones_list = bones_list
```

```
GL.x_bones = x_bones
```

setupBonesLists(hand):

En esta función se extraen los nombres de los huesos de la escena, se comparan con la lista de no permitidos que contiene aquellos que no tendrán hilo asociado, bien porque su rotación sea heredada del padre o porque no roten. Los añade todos a una lista y la añade a su vez a GL para su posterior uso en otras partes del script.

calibration():

Esta función se encarga de gestionar la recepción de la calibración. Es una función opcional ya que no se ha desarrollado un uso para esa calibración, por lo que está implementada como ejemplo de conexión a los sockets de transmisión de la calibración.

Abre una conexión con los sockets TCP/UNIX creados por `cal_sub.py`. Esta conexión se toma por parte del servidor que funciona en ese script como una petición de transmisión por lo que se reciben los datos de la calibración y se deserializan.

setupSockets():

Esta es una de las funciones más importantes. Es la encargada de gestionar la creación de los sockets en el arranque de la aplicación como se expuso en el apartado 3.3.4 Programación del Interfaz - Sockets.

fill_luts():

Finalmente, esta función se encarga de en la primera iteración rellenar dos Look-Up Tables. Una LUT es una tabla ordenada con pares clave-valor que se usa para ahorrar cálculos. Al inicio de la ejecución se rellenan con los valores resultantes de un cálculo complejo, en este caso la conversión del porcentaje transmitido por dglove a radianes para su aplicación en el guante. Así, cuando se inicia el funcionamiento de la aplicación en lugar de tener que hacer el cálculo cada vez que se recibe un dato, simplemente se hace un acceso a la tabla, que dispone de algoritmos de búsqueda optimizados.

En este caso se crean dos LUTs, una con ángulos de 90º para la flexión de los dedos y otra con ángulos de 45º para la expansión de los dedos.

```
lut1 = {x/10000 : round(-1*x/10000 * math.radians(90), 4) for
x in range (0, 10001)}

GL.lut1 = lut1

lut2 = {x/10000 : round(1*x/10000 * math.radians(45), 4) for x
in range (0, 10001)}

GL.lut2 = lut2
```

2.4 Hito 4

En esta sección se expone el desarrollo del Hito 4, que consiste en la creación de un interfaz gráfico de usuario (GUI) para el arranque y parada de los programas que forman parte de este Proyecto.

2.4.1 Motivación

La motivación para la inclusión de este nuevo hito es clara. Para arrancar toda la comunicación de la que se dispone hay que compilar dos programas manualmente, con unos argumentos determinados, arrancar 'roscore', y lanzar cuatro o cinco programas más, dependiendo de si queremos calibración o no en el entorno de realidad virtual, y todo esto siguiendo un orden fijo. Hacer esto manualmente es un proceso laborioso, y hasta cierto punto técnico, ya que requiere navegar por el sistema de archivos y ejecutar comandos por terminal.

Con este hito se solucionan los problemas previamente indicados, ya que permite compilar, lanzar y terminar los procesos a un alto nivel a través de una botonera virtual creada ex professo para este Proyecto.

Como este hito se decidió añadir hacia el final del trabajo, se buscó una forma de programar GUIs que tuviera las siguientes características:

- Que fuera sencillo de desarrollar, dado que se trata de una herramienta y no un fin para el TFG.
- Que permitiera una salida por pantalla intuitiva y limpia.
- Que tuviese soporte completo en Python 3.X dado que este es el futuro de Python, y es interesante que los scripts desarrollados funcionen bajo este marco.

Con los requisitos fijados y tras hacer un estudio de las librerías disponibles, se ha decidido utilizar la librería 'tkinter', dado que la curva de aprendizaje es bastante rápida, el entorno que resulta es rápido y sencillo de entender, y está totalmente soportada en Python 3.X.

Tkinter

Tkinter es la librería estándar de Python para el desarrollo de GUI. Es una capa de programación orientada a objetos, conducida por eventos, que funciona de interfaz con Tcl/Tk.

Tcl: Tool Command Language es un lenguaje de programación cross-platform de propósito general.

Tk: es un toolkit para interfaces gráficos de usuario. Es el GUI estándar para Tcl y para otros muchos lenguajes de programación.

2.4.2 El interfaz

Como se ha indicado en la sección anterior, tkinter funciona orientada a objetos, y conducida por eventos. Básicamente, cada elemento en el interfaz es un objeto que instancia de una clase perteneciente a la librería tkinter, y que dispone de una serie de propiedades para poder darle las características deseadas. Cuando seleccionamos en el interfaz gráfico ese elemento, se lanza un método de la clase, que ejecuta el comando que nosotros hayamos programado.

Estos comandos son para nosotros funciones callback, que llaman a funciones launcher que ejecutan el comando en un proceso independiente. Esto es necesario ya que si los lanzamos todos desde el mismo proceso, fagocitan el proceso del interfaz gráfico que deja de funcionar hasta que termina el proceso lanzado.

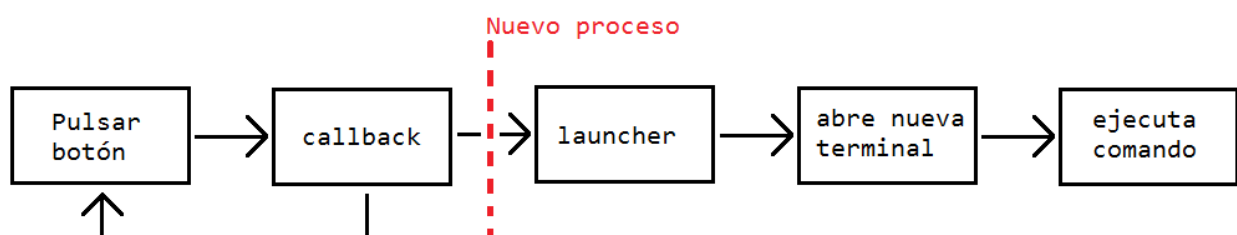


Figura 32: Proceso de ejecución de un comando desde el GUI

Dentro de los launcher, lo que se ha programado es sencillamente unos comandos de terminal que ejecutan en un nuevo terminal el proceso deseado.

Para ello se utiliza la función de Python `command` al que se le pasa un array con los diferentes argumentos del comando. En este caso, los dos primeros siempre son `xterm` para lanzar el nuevo terminal y `-e` como argumento para el anterior, para indicar que a continuación viene un comando que queremos ejecutar en este terminal creado. Pueden seguirlos uno o más comandos de bash. A continuación se muestra la lista completa de comandos programados. Todos tienen rutas relativas a la carpeta `scripts` del paquete entregado, en la que está situada la interfaz por defecto.

- `xterm -e sudo ../../../../devel/lib/tfg/dglove`
- `xterm -e ../../../../devel/lib/tfg/mano`
- `xterm -e python ./cal_sub.py`
- `xterm -e python ./main_sub.py`
- `xterm -e blender ./dgHand.blend`
- `xterm -e roscore`
- `xterm -e ./compilation.sh`

El último comando lanza un script en bash que ejecuta dos comandos, uno para compilar el paquete ros con `catkin_make`, y otro para compilar el programa `dGlove.cpp`.

Además, se han incluido una serie de imágenes para hacer más visual el entorno, así como mensajes de alerta cada vez que se lanza un comando, y un botón de terminación que, cuando es pulsado, termina el proceso principal y por consiguiente todos los demás (ver diferencia forking y spawning en 3.2.2 Relación y descripción de los ficheros generados - Creación de un objeto) mandando a este la señal `SIGTERM`.



Figura 33: Botonera GUI desarrollada con Tkinter

El orden de lanzamiento queda especificado en el manual de usuario que se encuentra en el Anexo I - Manual de Usuario.

2.5 Funcionamiento integrado

Hasta ahora se ha descrito como funciona cada parte de la aplicación por separado. En este apartado se expone el funcionamiento integrado del resultado de todos los hitos desde un punto de vista técnico. De cara al usuario, gran parte de los procesos aquí descritos son transparentes. Por ello se ha redactado un Manual de Usuario que se incluye como Anexo I - Manual de Usuario.

Este apartado toma como punto de partida la situación en que ya se tiene correctamente configurado el paquete y el entorno, y cada archivo está en su lugar.

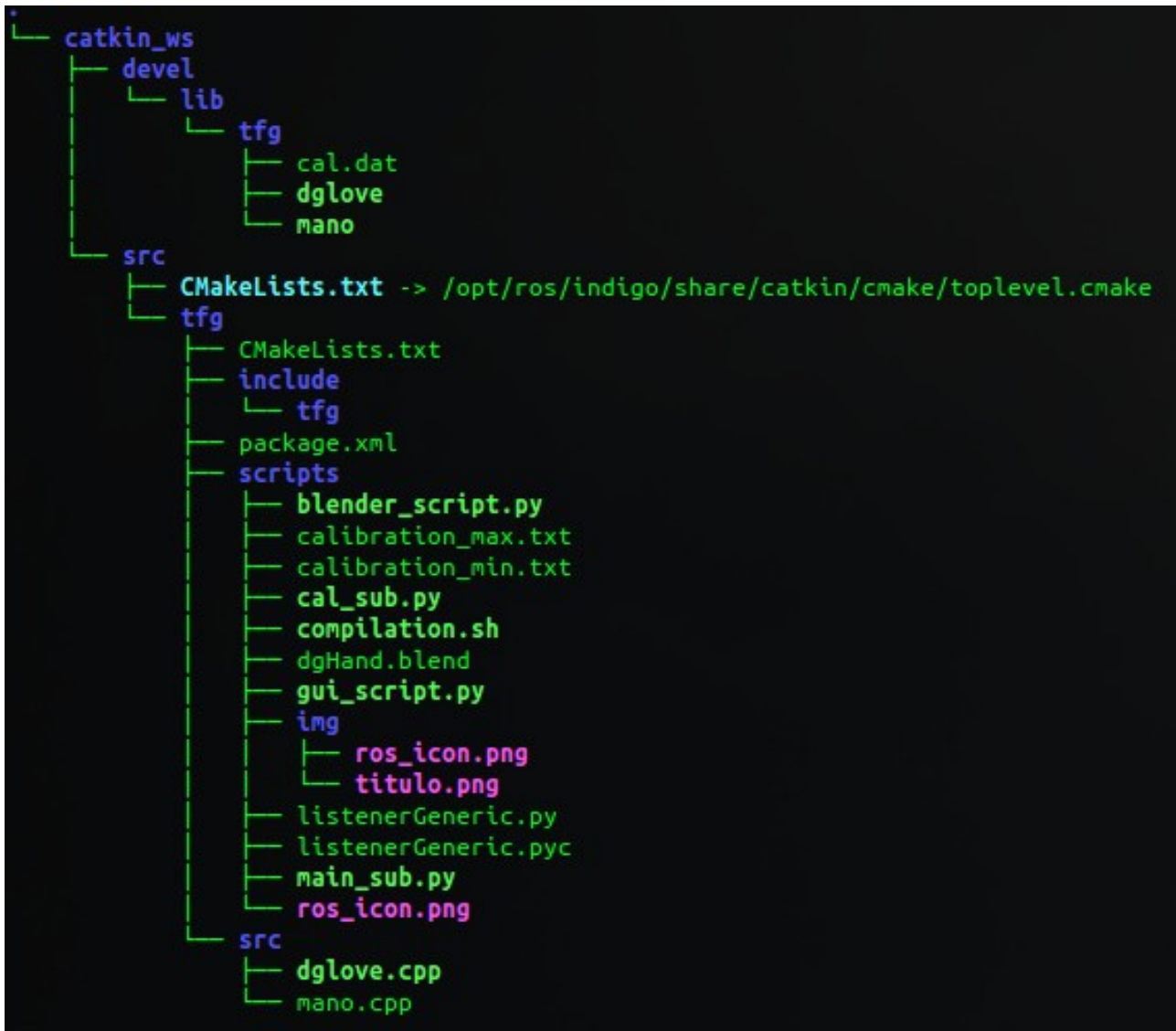


Figura 34: Árbol de directorios propios del proyecto

En la imagen anterior se muestra el árbol de directorios con los archivos y carpetas que forman parte de este proyecto (por claridad se han eliminado todos los que forman parte de la configuración básica de ROS).

2.5.1 Lanzamiento de la aplicación

Para que cada etapa de la comunicación lea y transmita los datos correctamente es necesario seguir un orden de lanzamiento. Gracias al interfaz de usuario la tarea de lanzar los procesos se ha visto muy simplificada. Para arrancar el interfaz se abre un terminal y se introduce el siguiente comando:

```
$ cd $HOME/nombre_directorio_ros/catkin_ws/src/tfg/scripts/
```

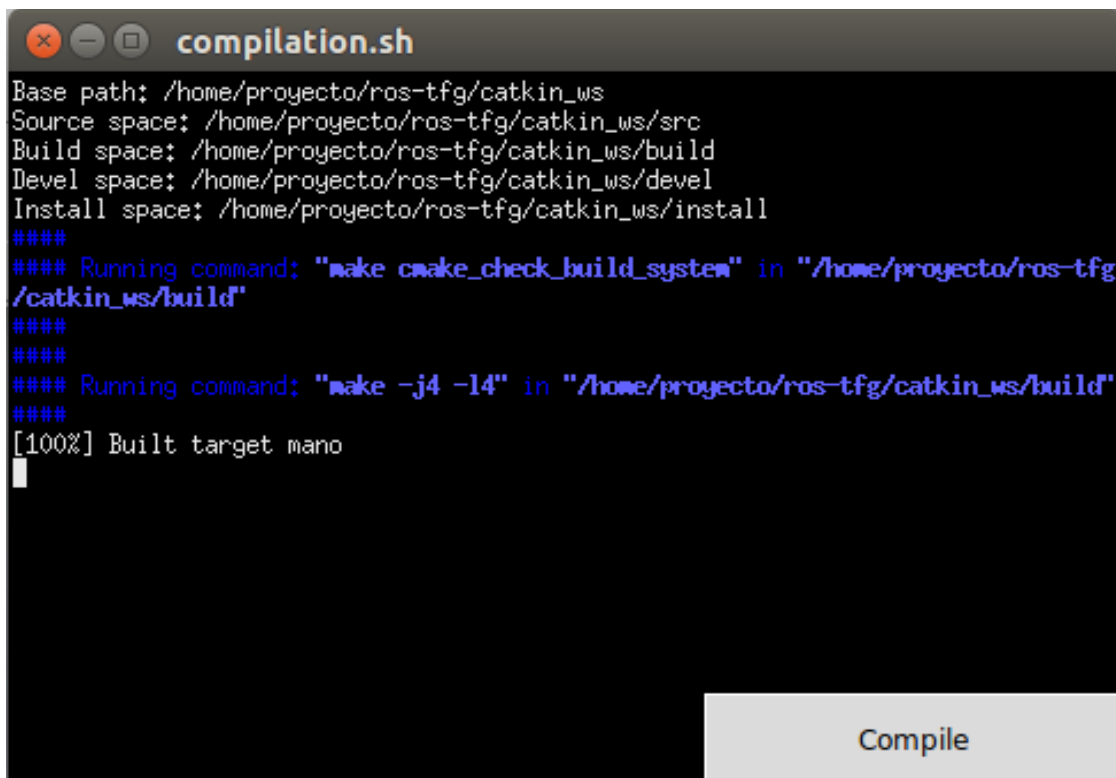
Donde nombre_directorio_ros se sustituye por el nombre que se le haya dado a la carpeta que contiene el workspace de catkin. En caso de haber seguido al pie de la letra los tutoriales mostrados en la página esta ruta no existirá porque la carpeta catkin_ws estará en la carpeta principal del usuario y el primer comando quedará como:

```
$ cd $HOME/catkin_ws/src/tfg/scripts
```

Cuando se esté en el directorio anterior, se lanza el interfaz con el siguiente comando:

```
$ python3 gui_script.py
```

Una vez lanzado el interfaz, usaremos este para compilar los dos programas escritos en C++. Para ello solo hay que pulsar e el botón que pone “Compilar” y se lanzará automáticamente una terminal en la que aparecerá el proceso de compilación en marcha, y que se cerrará sola una vez se haya completado.



```
compilation.sh
Base path: /home/proyecto/ros-tfg/catkin_ws
Source space: /home/proyecto/ros-tfg/catkin_ws/src
Build space: /home/proyecto/ros-tfg/catkin_ws/build
Devel space: /home/proyecto/ros-tfg/catkin_ws/devel
Install space: /home/proyecto/ros-tfg/catkin_ws/install
####
#### Running command: "make cmake_check_build_system" in "/home/proyecto/ros-tfg/catkin_ws/build"
####
####
#### Running command: "make -j4 -l4" in "/home/proyecto/ros-tfg/catkin_ws/build"
####
[100%] Built target mano
█
```

Figura 35: Terminal de compilación

Tras compilarlos, ha llegado el momento de lanzar los programas. Aunque el orden puede variar en algunos casos, es recomendable seguir siempre el aquí expuesto para evitar complicaciones.

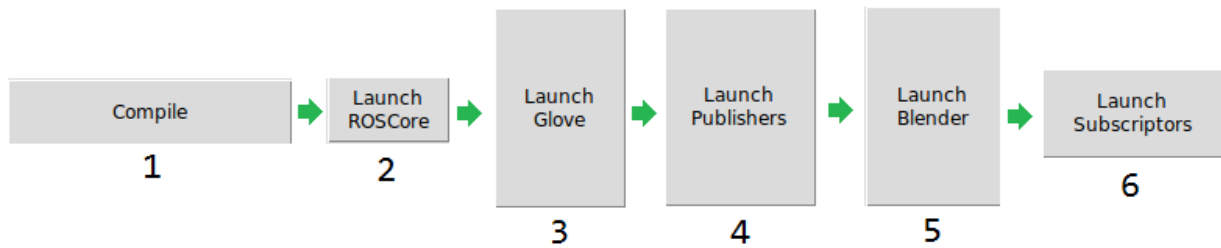


Figura 36: Proceso de arranque

Gracias al paradigma publicador/subscriptor que implementa ROS, tendremos dos secciones bien diferenciadas.

2.5.2 Etapa 1 - Publicación de los datos

Antes de comenzar a lanzar los programas propios, es necesario arrancar roscore, simplemente pulsando sobre el botón con este nombre.

```

roscore http://big-buddy:11311/
Press Ctrl-C to interrupt
Done checking log file disk usage, Usage is <1GB.

started roslaunch server http://big-buddy:38990/
ros_comm version 1.11.20

SUMMARY
=====

PARAMETERS
* /rostdistro: indigo
* /rosversion: 1.11.20

NODES

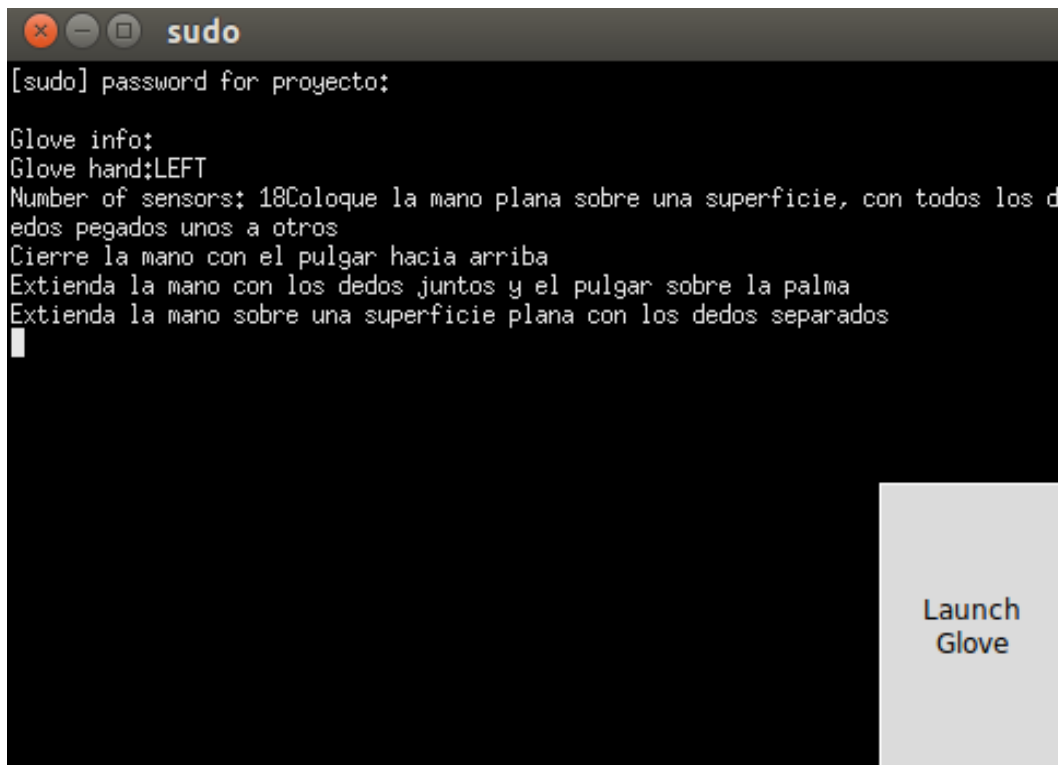
auto-starting new master
process[master]: started with pid [14370]
ROS_MASTER_URI=http://big-buddy:11311/

setting /run_id to 85919154-d766-11e6-b2b3-90004e9682fa
process[rosout-1]: started with pid [14383]
started core service [/rosout]

```

Figura 37: Terminal roscore

El primer paso es lanzar el programa dGlove. Este programa es el driver del guante, que lee los datos del mismo y configura los sockets de comunicación con el publicador en ROS. Al arrancarlo se abre un terminal que nos pide la contraseña, ya que son necesarios permisos de superusuario para acceder al puerto serial. Una vez introducida, se inicia el proceso de calibración del guante. Para realizarlo colocamos la mano en la pose indicada y se pulsa intro para cada una de las cuatro poses necesarias.



```
sudo
[sudo] password for proyecto:
Glove info:
Glove hand:LEFT
Number of sensors: 18Coloque la mano plana sobre una superficie, con todos los d
edos pegados unos a otros
Cierre la mano con el pulgar hacia arriba
Extienda la mano con los dedos juntos y el pulgar sobre la palma
Extienda la mano sobre una superficie plana con los dedos separados
█
```

Launch
Glove

Figura 38: Terminal mostrando el proceso de calibración

Una vez introducidas el programa inicia el comportamiento de servidor, por lo que queda a la espera de que se establezca una conexión TCP/IP por parte de un cliente. En este momento lanzamos el programa mano que inicia la comunicación con dGlove, lee la calibración, los primeros datos transmitidos e inicia la publicación en ROS. Esta sección es independiente de la segunda sección, ya que la publicación de los datos es independiente de la suscripción a los mismos.

```
sudo
^Sending MAX calibration data^
2279
2900
2722
2235
4093
3108
2627
2541
2573
2007
2143
2642
1309
2545
^Sending MIN calibration data^
Pulgar1:0.00,Pulgar2:0.00, Pulgarindice:0.00, indice1:0.06,indice2:0.00, indi
orazon:1.00, corazon1:0.00,corazon2:0.31, corazonanular:0.00, anular1:0.00, a
ar2:0.29, anularmenique:1.00, meñique1:0.00, meñique2:0.25,volumen: 2, count:
Sending data
Pulgar1:0.00,Pulgar2:0.00, Pulgarindice:0.00, indice1:0.06,indice2:0.00, indi
orazon:1.00, corazon1:0.00,corazon2:0.30, corazonanular:0.00, anular1:0.00, a
ar2:0.29, anularmenique:1.00, meñique1:0.00, meñique2:0.24,volumen: 2, count:
█
```

Figura 39: Terminal mandando la calibración

```
mano
cal_max[4]= 4092.000000
cal_max[5]= 3142.000000
cal_max[6]= 3145.000000
cal_max[7]= 3212.000000
cal_max[8]= 2417.000000
cal_max[9]= 2565.000000
cal_max[10]= 3231.000000
cal_max[11]= 2607.000000
cal_max[12]= 2350.000000
cal_max[13]= 3093.000000
cal_min[0]= 2279.000000
cal_min[1]= 2900.000000
cal_min[2]= 2722.000000
cal_min[3]= 2235.000000
cal_min[4]= 4093.000000
cal_min[5]= 3108.000000
cal_min[6]= 2627.000000
cal_min[7]= 2541.000000
cal_min[8]= 2573.000000
cal_min[9]= 2007.000000
cal_min[10]= 2143.000000
cal_min[11]= 2642.000000
cal_min[12]= 1309.000000
█
```

Figura 40: Terminal recibiendo la calibración

2.5.3 Etapa 2 – Suscripción a los datos y representación gráfica

Una vez esté la primera parte publicando datos en ROS, iniciaremos el lanzamiento de la segunda parte. En este caso se dispone de un script opcional, el de la calibración, que se puede lanzar o no dependiendo de si se desea tener los datos de la calibración en el entorno de realidad virtual. EN caso afirmativo, este debe ser el primer programa en lanzarse, ya que inicia un servidor TCP/UNIX que queda a la espera de una conexión del cliente.

Independientemente de si se ha lanzado este programa o no, el siguiente programa en lanzarse es el entorno de realidad virtual Blender, ya que al utilizar sockets UDP/UNIX para recibir los datos, es necesario que sea el programa que los va a leer quien los lance a diferencia del TCP. Esto es lógico teniendo en cuenta que UDP es un protocolo sin conexión, que se limita a enviar los datos a un socket creado previamente por el cliente.

Al abrirse Blender, actualmente está desactivado el autoarranque del Blender Game Engine, por lo que cuando se abra el programa aparecerá el entorno de desarrollo. Para entrar en el entorno de simulación tenemos varias opciones, pero lo más recomendable desde el punto de vista operacional es acceder al modo Blender Game Engine (deberíamos entrar en este modo por defecto cuando se arranque Blender desde el GUI) y en el menú de abajo a la derecha mostrado en la siguiente figura seleccionar “Start” en el modo que queramos, con monitor embebido o independiente.



Figura 41: Menú de lanzamiento del interfaz de Blender

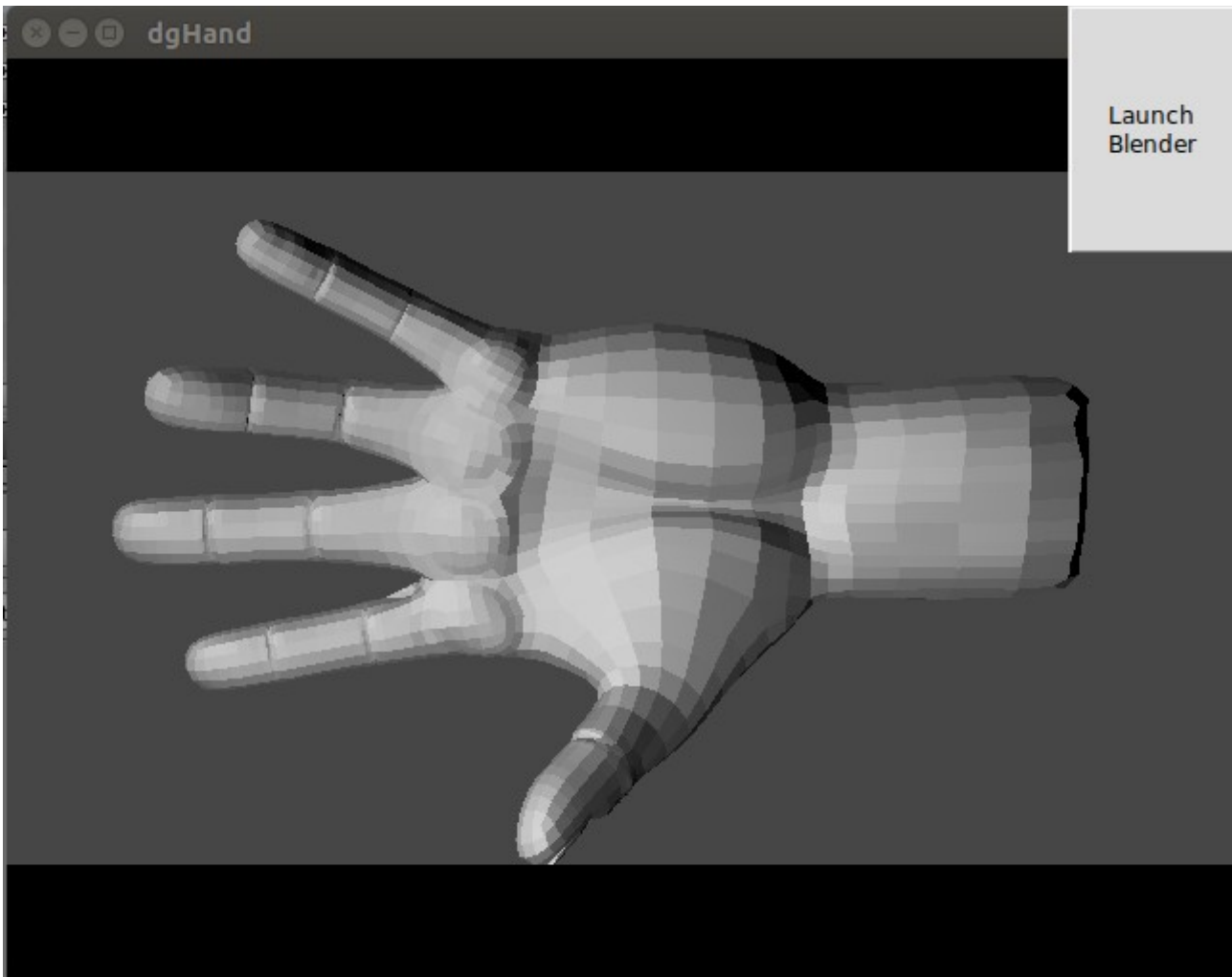


Figura 42: Mano renderizada en pose inicial

Una vez arrancado el entorno de realidad virtual, el último paso es lanzar el programa subscriptor de ROS para los datos, que lee de ROS y escribe en los sockets UDP los datos de funcionamiento del guante. En este momento estaría completamente arrancado y deberíamos ver el modelo de la mano simulando los movimientos de la mano que tiene el guante sensorizado puesto.

Finalmente se ha añadido un botón QUIT para terminar toda la aplicación, enviando una señal SIGTERM al proceso principal y todos los demás subprocesos.

3 Resultados

En este apartado se van a exponer resultados cuantitativos de velocidad de respuesta, uso de recursos y pérdida de datos por parte de la aplicación.

3.1 Velocidad de reacción

En este apartado se exponen los resultados de tiempos desde que un paquete es enviado por el primer proceso hasta que es implementado en el entorno de realidad virtual. Fundamentalmente es un estudio de la latencia de los paquetes.

Se parte de la base de que el guante se muestrea a una frecuencia de 10HZ, por lo que los slots de tiempo no pueden superar los 0,1s.

Utilizando el logger tanto de `main_sub.py` como de `blender_script.py`, y funciones de control de tiempo de C++ y Python se han recogido los siguientes resultados:

- Tiempo desde que sale un paquete hasta que se hace `hand.update()` con ese valor aplicado: 0,1005828s
- Tiempo de lanzamiento de los hilos: 0,0115144s
- Tiempo de iteración de Blender entre dos lanzamientos consecutivos sin leer datos: 0,0060699s
- Tiempo de iteración de Blender entre dos lanzamientos consecutivos habiendo leído datos: 0,0187547s

Como se puede ver por los resultados arrojados por las pruebas, el tiempo de iteración de Blender es más que de sobra para los tiempos fijados por la primera parte de la aplicación.

3.2 Uso de recursos

A continuación se exponen los resultados de una medición de recursos empleados por la aplicación y una breve discusión al respecto. Puesto que el

uso de recursos en ningún momento ha planteado un serio problema, nos limitaremos a analizar el cómputo global de recursos empleados en lugar de ir separando por hitos.

Con el sistema en reposo, se arranca el programa htop con el que se mide el uso de CPU y memoria no cacheada, y con el comando free vemos el consumo instantáneo de memoria desglosado.

```

1  [|||||] 8.1%
2  [||||] 4.0%
3  [|||||] 6.6%
4  [|||||] 7.9%
Mem[|||||] 753/7846MB
Swp[  ] 0/5008MB

Tasks: 115, 217 thr; 1 running
Load average: 1.09 2.50 1.68
Uptime: 00:14:08

```

Figura 43: Uso de CPU y memoria no cacheada en reposo

En la imagen podemos apreciar el bajo consumo que presenta el sistema en reposo, con solo funcionando dos programas a parte de los básicos de funcionamiento, htop y un dock de escritorio. La memoria mostrada es la memoria RAM utilizada y que no es liberable. Con la siguiente imagen se explica esto.

```

total      usado      libre      compart.  búffers  almac.
Memoria:  8034564  1759368  6275196  17996    187336  818412
-/+ buffers/cache:  753620  7280944
Swap:      5129212  0        5129212

```

Figura 44: Desglose de memoria en reposo

En la figura se puede ver el desglose de la cantidad de memoria utilizada en reposo. En la columna “usado” el valor de la primera fila se corresponde con la cantidad de memoria RAM utilizada en total. Para acelerar las ejecuciones, Linux almacena en memoria muchos elementos para ser recuperados más rápidamente, esto es el cacheo de información al que nos referíamos antes.

La segunda fila de esta columna se corresponde con el valor de la memoria usada sin cachear, es decir la memoria que está siendo usada realmente para las ejecuciones y que es indispensable que esté ocupada. La tercera columna es la Swap, que salta en caso de llenado de la RAM.

Con estos valores en reposo, pasamos a ejecutar la aplicación al completo con lo que se obtuvieron los siguientes datos.

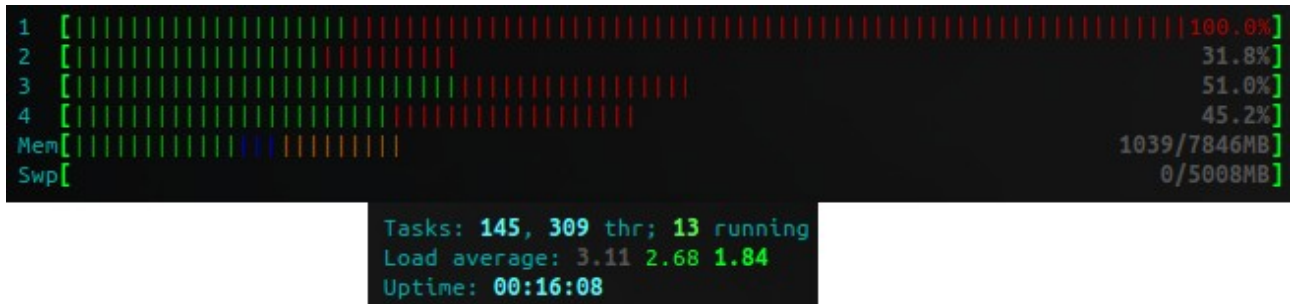


Figura 45: Uso de CPU y memoria no cacheada en funcionamiento

Sin lanzar ningún programa más que los pertenecientes a la aplicación, obtuvimos los incrementos de uso mostrados en la imagen. Se puede apreciar que el uso de CPU sí es bastante exigente, en parte debido a la cantidad de hilos que se lanzan (se puede ver el incremento absoluto en el recuadro inferior) ya que Blender hace un uso intensivo de ellos para el renderizado.

Por otro lado, se puede apreciar que el incremento de memoria consumida no es muy significativo. Esto se ve mejor con la siguiente imagen.

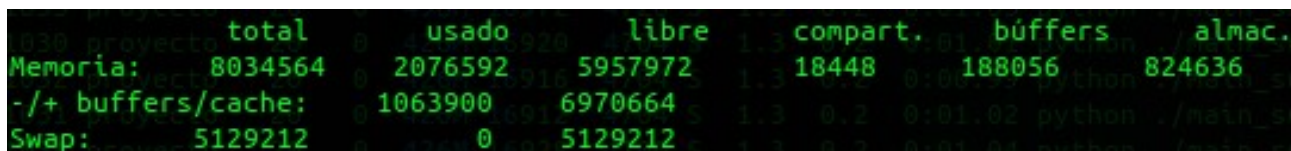


Figura 46: Desglose de memoria en funcionamiento

Como se puede apreciar, se ha pasado de un consumo de 753MB en reposo de memoria no cacheada a 1063MB, y de 1759MB de memoria RAM usada en total a 2076MB.

Viendo estos resultados se puede concluir que si bien es una aplicación exigente, no es inabordable, ya que estos resultados son fácilmente mejorables gracias al uso de un ordenador con una tarjeta gráfica mejor que permita descargar más trabajo de la CPU, así como una CPU con más núcleos. Sin embargo, hay que tener en cuenta que uno de los procesos que está corriendo, dglove, está planeado separarlo en otra máquina, por lo que bajará el uso de recursos.

3.3 Pérdida de información

En este apartado se estudia la posible pérdida de información debido a los sockets UNIX/UDP usados entre los Hitos 3 y 4 para comunicar los subscriptores de ROS con los hilos de Blender.

Para comprobar este apartado se incluyeron dos contadores, uno en la librería `listenerGeneric` y otro en `blender_script`. El primero es simplemente una variable que se importa de forma que todos los procesos pueden hacer uso de ella, cada vez que se manda un paquete se incrementa en una unidad esta variable `contador_paquetes`. Por el otro lado, en `blender_script` se ha añadido una variable a la persistencia de la misma forma que se hizo en las LUTs y los sockets (ver 3.3- Hito 3).

```
GL.paquetes_recibidos # Eliminada tras la prueba
```

Así se ha podido comprobar que no existe pérdida de paquetes. Esto es lo esperable ya que se podrían perder por tres razones:

1. Pérdida de los datos en la transmisión en red, pero como en este caso no hay transmisión en red, queda descartado.
2. Corrupción de memoria: es un hecho muy poco frecuente, que es posible que pase si se deja la aplicación corriendo el tiempo suficiente, pero no puede tener un efecto notable sobre el normal desarrollo.
3. Desbordamiento del socket: si el socket se encuentra lleno, al ir a escribir simplemente no habría hueco y ese paquete se desecharía. Si bien esta es la causa más posible de fallos, como se ha visto en el apartado 4.1- Velocidad de reacción, lo cierto es que hay margen de sobra en la velocidad de lectura de Blender de los sockets en relación con la velocidad de escritura del publicador.

4 Líneas futuras

El Proyecto hasta aquí presentado permite una serie de desarrollos y mejoras que podrían hacerse en un futuro para ampliar la funcionalidad y mejorar la existente.

Mejoras importantes de funcionalidad:

1. Mejora de la calibración del guante: a pesar de haber usado el procedimiento de calibración recomendado por la empresa, está claro que no da un resultado bueno. Para los sensores de flexión los datos que aporta son una aproximación suficiente pero para los de expansión está claro que falla. Una línea de investigación futura interesante sería el desarrollar un proceso de calibración mejor, ya que el hardware que se ha utilizado tiene la capacidad de proporcionar mejores datos.
2. Hacer el sistema más tolerante a fallos, porque si bien ahora es capaz de tratar excepciones en las partes críticas del código, no es posible subsanarlos ni hacer frente a fallos más complejos. De esta forma se plantean las siguientes mejoras:
 1. Hacer el sistema redundante: aumentar el BACKLOG de dglove.cpp a 2, de forma que se pueda lanzar otra aplicación paralela que entre en funcionamiento si se pierde la conexión entre el driver del guante y el programa publicador en ROS.
 2. Hacer un programa externo vigilante, que se encargue de comprobar si todos los procesos están vivos y respondiendo (checkeo "is alive?").
 3. En línea con lo anterior, cuando un proceso hijo o un hilo termina, envía al proceso padre o hilo principal una señal SIGCHD. Con un tratamiento de señal adecuado se puede utilizar esta señal para comprobar si el subproceso se ha caído, y en tal caso volver a lanzarlo.
3. Finalmente, el interfaz de Blender puede sufrir diversas mejoras:
 1. Mejorar la "weight paint", el valor de la influencia de cada hueso sobre la malla, de forma que el resultado sea más realista.
 2. Hacer un estudio detallado del movimiento de la mano para poder

representar con mayor fidelidad los movimientos, en especial el de expansión de los dedos.

3. Mejorar la articulación del dedo pulgar, ya que al tratarse de una articulación tipo bola realiza un movimiento compuesto.
4. Finalmente, como se dijo en el apartado 1.1.4 Marco del proyecto: Sistema robotizado colaborativo para cirugía laparoscópica asistida por la mano, este proyecto es solo una primera aproximación al objetivo final, por lo que a partir de este se debe hacer un gran proceso de integración con las demás partes del sistema que incluya:
 - La adecuación de los publicadores y suscriptores a las necesidades del sistema global.
 - Creación de un entorno para la mano virtual, a partir de datos de visión.

Ampliación de la funcionalidad de la aplicación:

1. Crear una biblioteca específica para el Hito 1, ya que actualmente se tienen varias funciones repetidas que se podrían añadir a otro archivo e importarlo de forma similar a lo hecho en el Hito 2.
2. Añadir `dglove.cpp` a `CmakeLists.txt` para compilar todo con el comando `catkin_make` en lugar de usar dos comandos independientes.
3. Ampliar la funcionalidad del GUI creado. La librería Tkinter incluye una gran variedad de objetos que pueden ser usados para pasar de un simple lanzador a un monitor de estado de la aplicación, similar a los usados en SCADAs.
 - Añadir unos checkboxes para seleccionar el nivel del Logger.
 - Añadir señales que indiquen:
 1. El estado de compilación (sin hacer, desactualizada, actualizada).
 2. El estado del sistema, consumo de recursos, latencia...
 3. Lanzamiento de avisos y señales de fallos.

5 Conclusiones

Los objetivos de este proyecto eran crear un modelo de una mano izquierda en un entorno de realidad virtual, y crear la capa de comunicación entre el guante sensorizado y la virtualización.

Con respecto al segundo apartado se han cumplido plenamente los objetivos ya que la comunicación es completa y rápida, con unos bajos tiempos de latencia y con un transporte íntegro y consistente de los datos emitidos.

De cara al segundo objetivo, el desarrollo se ha llevado a cabo y cumple con las especificaciones requeridas, pero es aún mejorable como se indica en el apartado 4. Líneas futuras. Sin embargo, gracias al repertorio de funciones que se han implementado tanto en el entorno Blender como en la librería creada para el Hito 3 y viendo los resultados, queda claro que mejorarlo es una cuestión de tiempo y ajustes, y no de falta de funcionalidad.

6 Bibliografía

- *Manetta, C. and Blade, R. (1995). "Glossary of VR Terminology". The International Journal of Virtual Reality.*
- *Azuma, Robert T. (1997). "A Survey of Augmented Reality". Presence: Teleoperators and virtual environments.*
- *Milgram, Paul; H. Takemura; A. Utsumi; F. Kishino (1994). "Augmented Reality: A class of displays on the reality-virtuality continuum". Proceedings of Telemanipulator and Telepresence Technologies*
- "Virtual Reality and Technologies for Combat Simulation". Office of technology assessments. Congress of the United States. (Accedido a través del repositorio "The OTA Legacy" mantenido por la Pinceton University).
- "Facebook to Acquire Oculus". Facebook Newsroom. Facebook. March 25, 2014 (Accedido el 12/12/2016).
- "ESTADO DEL ARTE Aplicación de nuevas tecnologías inmersivas para la innovación en la maquinaria de construcción, obras públicas y minería." Instituto Tecnológico de Aragón ITANOVA.
- *Anthes, C; García Hernández, R. J.;Wiedermann, M.; Kranzmüller, D. "State of the Art of Virtual Reality Technologies" 2016 IEEE Aerospace Conference, At Big Sky, Montana, United States*
- American Society of Colon and Rectal Surgeons. "Cirugía Laparoscópica" <https://www.fascrs.org/cirugia-laparoscopica> (Accedido el 12/12/2016).
- *Santos, L.; González, J.L.; Turiel, J.P.; Fraile, J.C.; de la Fuente, E. "Guante de datos sensorizado para uso en cirugía laparoscópica asistida por la mano (HALS)" ITAP - Instituto de las Tecnologías Avanzadas de la Producción, Universidad de Valladolid.*
- "5DT Data Glove Ultra Manual v1.3" Fifth Dimension Technologies
- "www.ros.org/core-components/" Accedido el 14/12/2016

- ["wiki.ros.org/catkin/conceptual_overview"](http://wiki.ros.org/catkin/conceptual_overview) Accedido el 14/12/2016
- ["www.kernel.org/pub/linux/utils/kernel/hotplug/udev/udev.html"](http://www.kernel.org/pub/linux/utils/kernel/hotplug/udev/udev.html)
Accedido el 16/12/2016
- *"Principles of Data Acquisition and Conversion"* Texas Instruments Application Report SBAA051A -January 1994 -Revised April 2015.
- ["wiki.ros.org/rospy"](http://wiki.ros.org/rospy)
 - ["wiki.ros.org/rospy_tutorials"](http://wiki.ros.org/rospy_tutorials) Accedido el 20/12/2016
 - ["wiki.ros.org/rospy/Overview/Publishers%20and%20Subscribers"](http://wiki.ros.org/rospy/Overview/Publishers%20and%20Subscribers)
Accedido el 20/12/2016
- *"Oracle9i Application Developer's Guide - 17 Using the Publish-Subscriber Model for Applications"* Release 2 (9.2) docs.oracle.com
- Márquez García, F. *"UNIX. Programación avanzada."* 3ª Ed. (2004)
- ["docs.python.org/2/tutorial/classes.html"](http://docs.python.org/2/tutorial/classes.html) Accedido el 22/12/2016
- Cook, William R.; Hill, Walter; Canning, Peter S. *"Inheritance is not subtyping."* (1990) Proc. 17th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages.
- Feneis, H.; Dauber W. *"Nomenclatura Anatómica Ilustrada"* (2014) 5ªEd
- Netter, F.H. *"Atlas de anatomía humana"* (2011) 5ªEd
- ["www.blendswap.com/blends/view/81285"](http://www.blendswap.com/blends/view/81285) Link al modelo de la mano.
Accedido el 11/12/2015
- ["creativecommons.org/licenses/by/3.0/"](http://creativecommons.org/licenses/by/3.0/) Licencia del modelo de la mano.
Accedido el 2/1/2017
- ["www.blender.org/manual/ko/rigging/"](http://www.blender.org/manual/ko/rigging/) Accedido el 5/3/2016
- ["docs.python.org/3.5/library/multiprocessing.html"](http://docs.python.org/3.5/library/multiprocessing.html) Accedido el 20/4/2016
- ["docs.python.org/3.5/library/multithreading.html"](http://docs.python.org/3.5/library/multithreading.html) Accedido el 20/4/2016

- ["docs.python.org/3.5/library/logging.html"](https://docs.python.org/3.5/library/logging.html) Accedido el 30/8/2016
- ["www.python.org/download/releases/3.0/"](https://www.python.org/download/releases/3.0/) Accedido el 4/1/2017
- ["docs.python.org/3/library/tk.html"](https://docs.python.org/3/library/tk.html) Accedido el 4/1/2017
- ["www.tcl.tk/"](http://www.tcl.tk/) Accedido el 5/1/2017
- ["creativecommons.org/licenses/by-sa/4.0/legalcode"](https://creativecommons.org/licenses/by-sa/4.0/legalcode) Licencia en el código.

Anexos

Anexo I - Manual de usuario

Puesta en marcha de la aplicación

A continuación se enumeran y explican brevemente los pasos para la puesta en marcha de la aplicación, tomando como punto de partida un sistema que ya está configurado adecuadamente para correrla. Para el proceso de construcción consultar la documentación de ROS adjunta en la bibliografía.

Paso 1: Abrir un terminal en el icono de terminal o pulsando

```
Ctrl + Alt + T
```

Paso 2: introducir el siguiente comando.

```
$ cd ~/ruta_workspace_catkin/catkin_ws/src/tfg/scripts &&  
python3 gui_script.py
```

4. Nota: Donde “*ruta_workspace_catkin*” es la ruta hasta el directorio de catkin. Si se ha seguido el procedimiento normal estará en la carpeta principal por lo que el comando será:

```
$cd ~/catkin_ws/src/tfg/scripts && python3 gui_script.py
```

Paso 3: compilar la aplicación pulsando en el botón “*Compile*”.

Paso 4: lanzar roscore pulsando en el botón “*roscore*”.

Paso 5: conectar el guante a un puerto USB.

Paso 6: lanzar el driver del guante pulsando sobre el botón “*Launch Glove*”.

Paso 7: introducir la contraseña del usuario en el terminal que se ha abierto y seguir los pasos de la calibración del guante.

Paso 8: lanzar los publicadores pulsando en el botón “*Launch Publishers*”.

- Nota: en este momento debemos ver cómo el guante y los publicadores se empiezan a comunicar, y un gran número de líneas con datos empezarán a escribirse en las terminales.

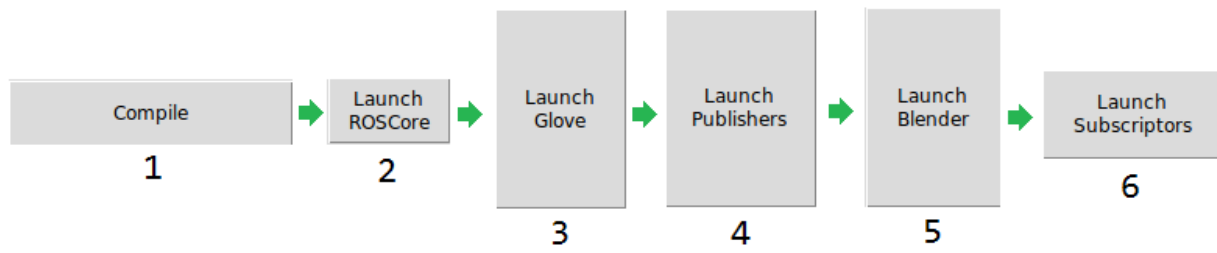
Paso 9: lanzar Blender pulsando en el botón Blender.

Paso 10: en el programa Blender, se pulsa sobre el botón un “*Start*” del menú que aparece en la imagen siguiente, o colocando el puntero sobre la pantalla que muestra el modelo de la mano, pulsar la tecla 'P'. El primero realiza la misma acción que pulsar 'P' que es iniciar un visor embebido en la pantalla en la que se muestra la mano. El segundo lo lanza en un visor independiente desde el punto de vista de la cámara colocada a ese propósito en la escena



- Nota: en este momento se debe ver la mano renderizada, el fondo pasa a negro y en el terminal de Blender empiezan a aparecer mensajes, sobretodo el mensaje “no data yet!”.

Paso 11: lanzar los subscriptores pulsando sobre el botón “*Launch Subscribers*”.



Anexo II – Código dglove.cpp

```
/**      dglove.cpp
Developed by Lidia Santos
and adapted by Pablo San José
        License CC-BY-SA      ***/

#include "fglove.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

#define MYPORT 3493      // Port for client's connections
#define BACKLOG 1      // Pending connections (enqueued)
#define DATA_LEN 14   // Length of the data vector
#define PKT_SIZE sizeof(struct packet) // Size of the packet

int sockfd; // Server socket (listening)
int newfd; // Connection socket, one for each client
```

```

struct sockaddr_in my_addr; // Server's IP and localport
struct sockaddr_in their_addr;// Client's IP and localport
socklen_t sin_size;// sizeof(sockaddr_in)

/* Structure for the packet */
struct packet{
    char type;
    float data[14];
};

/** error
Prints the given error and exits with the given status ***/
void error (char *errmsg, unsigned short errcode){
    perror(errmsg);
    exit(errcode);
}

/** reserve_d_pkt
Function that reserves memory for new packets PKT_SIZE ***/
packet* reserve_d_pkt(void){
    packet* new_pkt;
    new_pkt = (packet*)malloc(PKT_SIZE);

    if (new_pkt == NULL){
        printf("\nERROR AL RESERVAR MEMORIA PARA PKT\n");
    }
}

```

```

}

return(new_pkt);

}

/**
 * sendCalibration
 *
 * Sends first the values for the max calibration
 * and second the values for the min calibration
 */
void sendCalibration(unsigned short *cal_max, unsigned short *cal_min){

    printf("\nSending calibration data");

    packet *data_cal=NULL;

    data_cal = reserve_d_pkt();

    //Send data max
    data_cal->type = 'C'; // ASCII Code = 67

    for (int i = 0; i < DATA_LEN; i++){

        data_cal->data[i] = (float)cal_max[i];

        printf("\n%d", cal_max[i]);

    }

    if (send(newfd, data_cal, PKT_SIZE, 0) == -1){

        free(data_cal);

        error ((char *)"send", 2);

    }
}

```



```

else{
    printf("\n^Sending MAX calibration data^");
}

//Send data min
for (int i = 0; i < DATA_LEN; i++){
    data_cal->data[i] = (float)cal_min[i];
    printf("\n%d", cal_min[i]);
}

if (send(newfd, data_cal, PKT_SIZE, 0) == -1){
    free(data_cal);
    error((char *) "send", 2);
}
else{
    printf("\n^Sending MIN calibration data^");
}

free(data_cal);

}

/** accept_conn
Accepts new incoming connections */
void accept_conn(void){

    sin_size = sizeof(struct sockaddr_in);

```

```

    if ((newfd = accept(sockfd, (struct sockaddr *)&their_addr,&sin_size))
== -1) {

        error((char *)"accept", 1);

    }

    printf("server: conexion desde: %s\n", inet_ntoa(their_addr.sin_addr));

}

```

```

int main (int argc, char *argv[]) {

    fdGlove *pglove = NULL; //pointer to the glove

    float glovevalues[18]; //array to save the values

    unsigned short cal_data[18], cal_max[14], cal_min[14]; //arrays for the
calibration

    char* GLOVE_PORT = (char*) ("/dev/usb/hiddev0");

    int optval; // flag value for setsockopt

    packet *data = NULL; // Packet to be sent

    int count = 0; // counter

    int volumen;

    //Initialization of glove

    pglove = fdOpen(GLOVE_PORT);

    if (pglove == NULL) {

        printf("\nError. Glove not opened.\n");
    }
}

```

```

        return -1;
    }

//Glove Info

printf("\nGlove info:");

int glovetype = fdGetGloveType(pglove);

if (glovetype==FD_GLOVE5U_USB) {printf("\nGlove type:GLOVE5U_USB");}

int glovehand = fdGetGloveHand(pglove);

if (glovehand==FD_HAND_RIGHT) {printf("\nGlove hand:RIGHT");}

else {printf("\nGlove hand:LEFT");}

int numsensors = fdGetNumSensors(pglove);

printf("\nNumber of sensors: %d",numsensors);

//Calibration

printf("Coloque la mano plana sobre una superficie, con todos los dedos
pegados unos a otros");

getchar();

fdGetSensorRawAll(pglove,cal_data);

cal_min[2] = cal_data[2]; // Thumb/Index - Minimum Abduction
cal_min[3] = cal_data[3]; // Index Near - Minimum Flexure
cal_min[4] = cal_data[4]; // Index Far - Minimum Flexure
cal_min[5] = cal_data[5]; // Index/Middle - Minimum Abduction
cal_min[6] = cal_data[6]; // Middle Near - Minimum Flexure
cal_min[7] = cal_data[7]; // Middle Far - Minimum Flexure
cal_min[8] = cal_data[8]; // Middle/Ring - Minimum Abduction
cal_min[9] = cal_data[9]; // Ring Near - Minimum Flexure

```

```

cal_min[10] = cal_data[10]; // Ring Far - Minimum Flexure
cal_min[11] = cal_data[11]; // Ring/Little - Minimum Abduction
cal_min[12] = cal_data[12]; // Little Near - Minimum Flexure
cal_min[13] = cal_data[13]; // Little Far - Minimum Flexure

printf("Cierre la mano con el pulgar hacia arriba");

getchar();

fdGetSensorRawAll(pglove,cal_data);

cal_min[0] = cal_data[0]; // Thumb Near - Minimum Flexure
cal_min[1] = cal_data[1]; // Thumb Far - Minimum Flexure
cal_max[3] = cal_data[3]; // Index Near - Maximum Flexure
cal_max[4] = cal_data[4]; // Index Far - Maximum Flexure
cal_max[6] = cal_data[6]; // Middle Near - Maximum Flexure
cal_max[7] = cal_data[7]; // Middle Far - Maximum Flexure
cal_max[9] = cal_data[9]; // Ring Near - Maximum Flexure
cal_max[10] = cal_data[10]; // Ring Far - Maximum Flexure
cal_max[12] = cal_data[12]; // Little Near - Maximum Flexure
cal_max[13] = cal_data[13]; // Little Far - Maximum Flexure

printf("Extienda la mano con los dedos juntos y el pulgar sobre la
palma");

getchar();

fdGetSensorRawAll(pglove,cal_data);

cal_max[0] = cal_data[0]; // Thumb Near - Maximum Flexure
cal_max[1] = cal_data[1]; // Thumb Far - Maximum Flexure

```

```

    printf("Extienda la mano sobre una superficie plana con los dedos
separados");

    getchar();

    fdGetSensorRawAll(pglove, cal_data);

    cal_max[2] = cal_data[2]; // Thumb/Index - Maximum Abduction
    cal_max[5] = cal_data[5]; // Index/Middle - Maximum Abduction
    cal_max[8] = cal_data[8]; // Middle/Ring - Maximum Abduction
    cal_max[11] = cal_data[11]; // Ring/Little - Maximum Abduction

    fdSetCalibrationAll(pglove, cal_max, cal_min);

    //Socket setup

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1){
        error((char *)"socket", 1);
    }

    /* setsockopt: Handy debugging trick that lets
    * us rerun the server immediately after we kill it;
    * otherwise we have to wait about 20 secs.
    * Eliminates "ERROR on binding: Address already in use" error.
    */

    optval = 1;

    setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR,
        (const void *)&optval , sizeof(int));

```

```

my_addr.sin_family = AF_INET; //IP v4 protocol
my_addr.sin_port = htons(MYPORT); //assigns the port
my_addr.sin_addr.s_addr = INADDR_ANY; // local IP
bzero(&(my_addr.sin_zero), 8); // fills with zeros the structure

if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr))
== -1){
    error((char *)"bind", 1);
}

if (listen(sockfd, BACKLOG) == -1){
    error((char *)"listen", 1);
}

// Accept incoming connections
accept_conn();

// Send calibration
sendCalibration(cal_max, cal_min);

// Send data
packet* data_send = NULL;
int i;

data_send = reserve_d_pkt();

```

```

//Header for data

data_send->type = 'D';

while(1){

    fdGetSensorScaledAll (pglove,glovevalues);

    if (glovevalues[FD_THUMBNEAR]>0.5 || glovevalues[FD_THUMBFAR]>0.5 ||
glovevalues[FD_INDEXNEAR]>0.5 || glovevalues[FD_INDEXFAR]>0.5 ||
glovevalues[FD_MIDDLENEAR]>0.5 || glovevalues[FD_MIDDLEFAR]>0.5 ||
glovevalues[FD_RINGNEAR]>0.5 || glovevalues[FD_RINGFAR]>0.5 ||
glovevalues[FD_LITTLENEAR]>0.5 || glovevalues[FD_LITTLEFAR]>0.5)

        {volumen=1;}

    else

        {volumen=2;}

    // Skipping this line increases the speed

        printf("\nPulgar1:%.2f,Pulgar2:%.2f, Pulgarindice:%.2f,
indice1:%.2f,indice2:%.2f, indicecorazon:%.2f,
corazon1:%.2f,corazon2:%.2f, corazonanular:%.2f, anular1:%.2f,
anular2:%.2f, anularmenique:%.2f, meñique1:%.2f, meñique2:%.2f,volumen:
%d, count:
%d",glovevalues[FD_THUMBNEAR],glovevalues[FD_THUMBFAR],glovevalues[FD_THU
MBINDEX],glovevalues[FD_INDEXNEAR],glovevalues[FD_INDEXFAR],glovevalues[F
D_INDEXMIDDLE],glovevalues[FD_MIDDLENEAR],glovevalues[FD_MIDDLEFAR],glove
values[FD_MIDDLERING],glovevalues[FD_RINGNEAR],glovevalues[FD_RINGFAR],gl
ovevalues[FD_RINGLITTLE],glovevalues[FD_LITTLENEAR],glovevalues[FD_LITTLE
FAR],volumen, count);

    //Composition of the packet

    data_send->data[0]=glovevalues[FD_THUMBNEAR];

    data_send->data[1]=glovevalues[FD_THUMBFAR];

    data_send->data[2]=glovevalues[FD_THUMBINDEX];

    data_send->data[3]=glovevalues[FD_INDEXNEAR];

```

```

data_send->data[4]=glovevalues[FD_INDEXFAR];
data_send->data[5]=glovevalues[FD_INDEXMIDDLE];
data_send->data[6]=glovevalues[FD_MIDDLENEAR];
data_send->data[7]=glovevalues[FD_MIDDLEFAR];
data_send->data[8]=glovevalues[FD_MIDDLERING];
data_send->data[9]=glovevalues[FD_RINGNEAR];
data_send->data[10]=glovevalues[FD_RINGFAR];
data_send->data[11]=glovevalues[FD_RINGLITTLE];
data_send->data[12]=glovevalues[FD_LITTLENEAR];
data_send->data[13]=glovevalues[FD_LITTLEFAR];

if (send(newfd, data_send, PKT_SIZE, 0) == -1){
    error((char *)"send", 3);
}
else{
    printf("\nSending data");

}

count++;

usleep(100000);

}

free(data_send);

close(newfd);

```



```
//Closing glove

int closeg = fdClose(pglove);

if (closeg!=0)    {

    printf("\nGlove closed.\n\n");

}

else {

    printf("\nError closing glove.\n");

    return -1;

}

return 0;

}
```

Anexo III – Código

mano.cpp

```
/**      mano.cpp
Developed by Lidia Santos
and adapted by Pablo San José
License CC-BY-SA      ***/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <iostream>
#include <fstream>
#include <sys/wait.h>
#include "ros/ros.h"
#include <std_msgs/Float64.h>
#include <std_msgs/String.h>
#include <std_msgs/MultiArrayDimension.h>
#include <std_msgs/Float64MultiArray.h>
```

```

#define PORT 3493 // puerto al que vamos a conectar
#define DATA_LEN 14 // length of the data vector
#define DATA_SIZE 14*sizeof(float) // size of data vector
#define PKT_SIZE sizeof(struct packet) // size of the packet struct
#define SOCKADDR_IN_SIZE sizeof(struct sockaddr_in)
#define SOCKADDR_SIZE sizeof(struct sockaddr)

using namespace std;

int sockfd, numbytes;

struct hostent *he;

struct sockaddr_in their_addr; // informacion de la direccion de destino

/* Structure for the packet */
struct packet{
    char type;
    float data[14];
};

/** error
Prints the given error and exits with the given status */
void error (char *errmsg, unsigned short errcode){
    perror(errmsg);
    exit(errcode);
}

```

```
}
```

```
/** reserve_d_pkt
```

```
Function that reserves memory for new packets PKT_SIZE ***/
```

```
packet* reserve_d_pkt(void){
```

```
    packet* new_pkt;
```

```
    new_pkt = (packet*)malloc(PKT_SIZE);
```

```
    if (new_pkt == NULL){
```

```
        printf("\nERROR AL RESERVAR MEMORIA PARA PKT\n");
```

```
    }
```

```
    return(new_pkt);
```

```
}
```

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

```
    packet *data_recv = NULL; // pointer to reserved memory for struct
```

```
    int counter=0;
```

```
    float fcal_max[14];
```

```
    float fcal_min[14];
```

```
    //hostname
```

```
    if ((he=gethostbyname("127.0.0.1")) == NULL){
```

```

    error((char *)"gethostbyname", 1);
}

//Create socket
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1){
    error ((char *)"socket", 1);
}

their_addr.sin_family = AF_INET;
their_addr.sin_port = htons(PORT);
their_addr.sin_addr = *((struct in_addr *)he->h_addr);
memset(&(their_addr.sin_zero), 8, SOCKADDR_IN_SIZE);

if (connect(sockfd, (struct sockaddr *)&their_addr, SOCKADDR_SIZE) ==
-1){
    error ((char *)"connect", 1);
}

// Setting up ROS connection and publishers
ros::init(argc, argv, "glove");
ros::NodeHandle nh;

    ros::Publisher      calibration_max      =
nh.advertise<std_msgs::Float64MultiArray>("calibration_max",1);

    ros::Publisher      calibration_min      =
nh.advertise<std_msgs::Float64MultiArray>("calibration_min",1);

```

```

    ros::Publisher indice1 = nh.advertise<std_msgs::Float64>("indice1",
100);

    ros::Publisher indice2 = nh.advertise<std_msgs::Float64>("indice2",
100);

    ros::Publisher corazon1 = nh.advertise<std_msgs::Float64>("corazon1",
100);

    ros::Publisher corazon2 = nh.advertise<std_msgs::Float64>("corazon2",
100);

    ros::Publisher anular1 = nh.advertise<std_msgs::Float64>("anular1",
100);

    ros::Publisher anular2 = nh.advertise<std_msgs::Float64>("anular2",
100);

    ros::Publisher menique1 = nh.advertise<std_msgs::Float64>("menique1",
100);

    ros::Publisher menique2 = nh.advertise<std_msgs::Float64>("menique2",
100);

    ros::Publisher pulgar1 = nh.advertise<std_msgs::Float64>("pulgar1",
100);

    ros::Publisher pulgar2 = nh.advertise<std_msgs::Float64>("pulgar2",
100);

    ros::Publisher expansion =
nh.advertise<std_msgs::Float64MultiArray>("expansion", 100);

    std_msgs::Float64MultiArray str_cal_max;

    std_msgs::Float64MultiArray str_cal_min;

    std_msgs::Float64 str_msgind1;

    std_msgs::Float64 str_msgind2;

    std_msgs::Float64 str_msgcor1;

    std_msgs::Float64 str_msgcor2;

    std_msgs::Float64 str_msganu1;

    std_msgs::Float64 str_msganu2;

```

```

std_msgs::Float64 str_msgmen1;
std_msgs::Float64 str_msgmen2;
std_msgs::Float64 str_msgpull1;
std_msgs::Float64 str_msgpul2;
std_msgs::Float64MultiArray str_msgexp;

ros::Rate loop_rate(10);

int i=0;

data_recv = reserve_d_pkt();

while (ros::ok()){

    if ((numbytes=recv(sockfd, data_recv, PKT_SIZE, 0)) == -1) {
        error ((char *)"recv", 3);
    }

    // In case it's a calibration packet
    if(data_recv->type == 'C'){

        printf("\nCalibration mode");

        // prints the first packet
        for (i=0; i<DATA_LEN; i++){

            fcal_max[i] = data_recv->data[i];

            printf("\ncal_max[%d]= %f", i, fcal_max[i]);

        }

    }

}

/***/      mano.cpp

```

Developed by Lidia Santos

and adapted by Pablo San José

License CC-BY-SA ***/

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <iostream>
#include <fstream>
#include <sys/wait.h>
#include "ros/ros.h"
#include <std_msgs/Float64.h>
#include <std_msgs/String.h>
#include <std_msgs/MultiArrayDimension.h>
#include <std_msgs/Float64MultiArray.h>

#define PORT 3493 // puerto al que vamos a conectar
#define DATA_LEN 14 // length of the data vector
#define DATA_SIZE 14*sizeof(float) // size of data vector
```



```

#define PKT_SIZE sizeof(struct packet) // size of the packet struct

#define SOCKADDR_IN_SIZE sizeof(struct sockaddr_in)

#define SOCKADDR_SIZE sizeof(struct sockaddr)

using namespace std;

int sockfd, numbytes;

struct hostent *he;

struct sockaddr_in their_addr; // informacion de la direccion de destino

/* Structure for the packet */

struct packet{

    char type;

    float data[14];

};

/** error

Prints the given error and exits with the given status ***/

void error (char *errmsg, unsigned short errcode){

    perror(errmsg);

    exit(errcode);

}

/** reserve_d_pkt

Function that reserves memory for new packets PKT_SIZE ***/

packet* reserve_d_pkt(void){

```

```

packet* new_pkt;

new_pkt = (packet*)malloc(PKT_SIZE);

if (new_pkt == NULL){

    printf("\nERROR AL RESERVAR MEMORIA PARA PKT\n");

}

return(new_pkt);
}

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

    packet *data_recv = NULL; // pointer to reserved memory for struct

    int counter=0;

    float fcal_max[14];

    float fcal_min[14];

    //hostname

    if ((he=gethostbyname("127.0.0.1")) == NULL){

        error((char *)"gethostbyname", 1);

    }

    //Create socket

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1){

```

```

        error ((char *)"socket", 1);
    }

    their_addr.sin_family = AF_INET;
    their_addr.sin_port = htons(PORT);
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    memset(&(their_addr.sin_zero), 8, SOCKADDR_IN_SIZE);

    if (connect(sockfd, (struct sockaddr *)&their_addr, SOCKADDR_SIZE) ==
-1){
        error ((char *)"connect", 1);
    }

    // Setting up ROS connection and publishers

    ros::init(argc, argv, "glove");

    ros::NodeHandle nh;

        ros::Publisher      calibration_max      =
nh.advertise<std_msgs::Float64MultiArray>("calibration_max",1);

        ros::Publisher      calibration_min      =
nh.advertise<std_msgs::Float64MultiArray>("calibration_min",1);

    ros::Publisher indicel = nh.advertise<std_msgs::Float64>("indicel",
100);

    ros::Publisher indice2 = nh.advertise<std_msgs::Float64>("indice2",
100);

    ros::Publisher corazon1 = nh.advertise<std_msgs::Float64>("corazon1",
100);

    ros::Publisher corazon2 = nh.advertise<std_msgs::Float64>("corazon2",

```

```

100);

    ros::Publisher anular1 = nh.advertise<std_msgs::Float64>("anular1",
100);

    ros::Publisher anular2 = nh.advertise<std_msgs::Float64>("anular2",
100);

    ros::Publisher menique1 = nh.advertise<std_msgs::Float64>("menique1",
100);

    ros::Publisher menique2 = nh.advertise<std_msgs::Float64>("menique2",
100);

    ros::Publisher pulgar1 = nh.advertise<std_msgs::Float64>("pulgar1",
100);

    ros::Publisher pulgar2 = nh.advertise<std_msgs::Float64>("pulgar2",
100);

                                ros::Publisher      expansion      =
nh.advertise<std_msgs::Float64MultiArray>("expansion", 100);

    std_msgs::Float64MultiArray str_cal_max;

    std_msgs::Float64MultiArray str_cal_min;

    std_msgs::Float64 str_msgind1;

    std_msgs::Float64 str_msgind2;

    std_msgs::Float64 str_msgcor1;

    std_msgs::Float64 str_msgcor2;

    std_msgs::Float64 str_msganu1;

    std_msgs::Float64 str_msganu2;

    std_msgs::Float64 str_msgmen1;

    std_msgs::Float64 str_msgmen2;

    std_msgs::Float64 str_msgpull1;

    std_msgs::Float64 str_msgpul2;

    std_msgs::Float64MultiArray str_msgexp;

```

```

ros::Rate loop_rate(10);

int i=0;

data_recv = reserve_d_pkt();

while (ros::ok()){

    if ((numbytes=recv(sockfd, data_recv, PKT_SIZE, 0)) == -1) {

        error ((char *)"recv", 3);

    }

    // In case it's a calibration packet
    if(data_recv->type == 'C'){

        printf("\nCalibration mode");

        // prints the first packet
        for (i=0; i<DATA_LEN; i++){

            fcal_max[i] = data_recv->data[i];

            printf("\ncal_max[%d]= %f", i, fcal_max[i]);

        }

        // Reads the second packet
        if ((numbytes=recv(sockfd, data_recv, PKT_SIZE, 0)) == -1) {

            error ((char *)"recv", 2);

        }
    }
}

```

```

// prints the second packet

for (i=0; i<DATA_LEN; i++){

    fcal_min[i] = data_recv->data[i];

    printf("\ncal_min[%d]= %f", i, fcal_min[i]);

}

// creates the messages for ROS

str_cal_max.data.clear();

str_cal_min.data.clear();

for (i=0; i<14; i++){

    str_cal_max.data.push_back(fcal_max[i]);

    str_cal_min.data.push_back(fcal_min[i]);

}

} else{

    // this line can be skipped for speed

    printf("\nPulgar1: %.2f,Pulgar2: %.2f, Pulgarindice:
%.2f,indice1: %.2f,indice2: %.2f,indicemedio: %.2f, medio1: %.2f,medio2:
%.2f, medioanular: %.2f, anular1: %.2f,anular2: %.2f,anularmenique: %.2f,
meñique1: %.2f,meñique2: %.2f, bucle:%d",data_recv->data[0],data_recv-
>data[1],data_recv->data[2],data_recv->data[3],data_recv-
>data[4],data_recv->data[5],data_recv->data[6],data_recv-
>data[7],data_recv->data[8],data_recv->data[9],data_recv-
>data[10],data_recv->data[11],data_recv->data[12],data_recv->data[13],i);

    // creation of the messages for ROS

    str_msgpull1.data = data_recv->data[0];

    pulgar1.publish(str_msgpull1);

```

```
str_msgpul2.data = data_recv->data[1];
pulgar2.publish(str_msgpul2);

str_msgind1.data = data_recv->data[3];
indice1.publish(str_msgind1);

str_msgind2.data = data_recv->data[4];
indice2.publish(str_msgind2);

str_msgcor1.data = data_recv->data[6];
corazon1.publish(str_msgcor1);

str_msgcor2.data = data_recv->data[7];
corazon2.publish(str_msgcor2);

str_msganu1.data = data_recv->data[9];
anular1.publish(str_msganu1);

str_msganu2.data = data_recv->data[10];
anular2.publish(str_msganu2);

str_msgmen1.data = data_recv->data[12];
menique1.publish(str_msgmen1 );

str_msgmen2.data = data_recv->data[13];
menique2.publish(str_msgmen2);
```

```

// Publish sensors between fingers

str_msgexp.data.clear();

str_msgexp.data.push_back(data_rcv->data[2]);

str_msgexp.data.push_back(data_rcv->data[5]);

str_msgexp.data.push_back(data_rcv->data[8]);

str_msgexp.data.push_back(data_rcv->data[11]);

expansion.publish(str_msgexp);

// Publish calibration

calibration_max.publish(str_cal_max);

calibration_min.publish(str_cal_min);
}

ros::spinOnce();

loop_rate.sleep();

counter++;
}

free(data_rcv);

close(sockfd);

```



```

return 0;
}

// Reads the second packet
if ((numbytes=recv(sockfd, data_recv, PKT_SIZE, 0)) == -1) {
    error ((char *)"recv", 2);
}

// prints the second packet
for (i=0; i<DATA_LEN; i++){
    fcal_min[i] = data_recv->data[i];
    printf("\ncal_min[%d]= %f", i, fcal_min[i]);
}

// creates the messages for ROS
str_cal_max.data.clear();
str_cal_min.data.clear();
for (i=0; i<14; i++){
    str_cal_max.data.push_back(fcal_max[i]);
    str_cal_min.data.push_back(fcal_min[i]);
}

} else{

// this line can be skipped for speed

    printf("\nPulgar1: %.2f,Pulgar2: %.2f, Pulgarindice:
%.2f,indice1: %.2f,indice2: %.2f,indicemedio: %.2f, medio1: %.2f,medio2:
%.2f, medioanular: %.2f, anular1: %.2f,anular2: %.2f,anularmenique: %.2f,
meñique1: %.2f,meñique2: %.2f, bucle:%d",data_recv->data[0],data_recv-

```

```
>data[1],data_recv->data[2],data_recv->data[3],data_recv-  
>data[4],data_recv->data[5],data_recv->data[6],data_recv-  
>data[7],data_recv->data[8],data_recv->data[9],data_recv-  
>data[10],data_recv->data[11],data_recv->data[12],data_recv->data[13],i);
```

```
// creation of the messages for ROS
```

```
str_msgpull1.data = data_recv->data[0];
```

```
pulgar1.publish(str_msgpull1);
```

```
str_msgpul2.data = data_recv->data[1];
```

```
pulgar2.publish(str_msgpul2);
```

```
str_msgind1.data = data_recv->data[3];
```

```
indice1.publish(str_msgind1);
```

```
str_msgind2.data = data_recv->data[4];
```

```
indice2.publish(str_msgind2);
```

```
str_msgcor1.data = data_recv->data[6];
```

```
corazon1.publish(str_msgcor1);
```

```
str_msgcor2.data = data_recv->data[7];
```

```
corazon2.publish(str_msgcor2);
```

```
str_msganu1.data = data_recv->data[9];
```

```
anular1.publish(str_msganu1);
```

```
str_msganu2.data = data_recv->data[10];
```

```

anular2.publish(str_msganu2);

str_msgmen1.data = data_recv->data[12];
menique1.publish(str_msgmen1 );

str_msgmen2.data = data_recv->data[13];
menique2.publish(str_msgmen2);

// Publish sensors between fingers
str_msgexp.data.clear();
str_msgexp.data.push_back(data_recv->data[2]);
str_msgexp.data.push_back(data_recv->data[5]);
str_msgexp.data.push_back(data_recv->data[8]);
str_msgexp.data.push_back(data_recv->data[11]);

expansion.publish(str_msgexp);

// Publish calibration
calibration_max.publish(str_cal_max);
calibration_min.publish(str_cal_min);
}

ros::spinOnce();

loop_rate.sleep();

```

```
        counter++;  
    }  
  
    free(data_recv);  
  
    close(sockfd);  
  
    return 0;  
}
```


Anexo IV – Código listenerGeneric.py

```
#      main_sub.py
# Developed by Pablo San José
#      License CC-BY-SA      ***/

#!/usr/bin/python

from multiprocessing import Process

import rospy

import json

from std_msgs.msg import Float64

from std_msgs.msg import String

from std_msgs.msg import Float64MultiArray

import socket

import os

import sys

import threading

from array import array

import logging

import time

#####      Global variables      #####
```

```

address = '/tmp/' # Path for the sockets

# List of chat's names for flexion sensors
nl_fingers = ["indice1", "indice2",
              "corazon1", "corazon2",
              "anular1", "anular2",
              "menique1", "menique2",
              "pulgar1", "pulgar2"]

# List of chat's names for separation sensors
nl_sep = ["pulgarindice", "indicecorazon",
          "corazonanular", "anularmenique"]

#####      End Global variables      #####

#####      Listeners' classes      #####

class listenerGeneric():

    def __init__(self, name, id_p, socket=None):
        self.name = name
        self.id = id_p
        self.address = address + name
        if socket is None:

```

```

        socket = ""

    self.socket = socket

def run(self):

    logging.info(self.name + "START"+ str(time.time()))

    # Starts ROS communication

    self.listen()

    logging.info(self.name + "FINISH"+ str(time.time()))

class listenerFingers(listenerGeneric):

def listen(self):

    logging.info(self.name + " listen " + "START"+ str(time.time()))

    node_name = self.name + "_listener"

    rospy.init_node(node_name, anonymous=True)

    rospy.Subscriber(self.name, Float64, self.callback)

    logging.info(self.name + " spin " + str(time.time()))

    rospy.spin()

def callback(self, data):

        logging.info(self.name + " callback " + "START"+
str(time.time()))

        data_send = str(round(data.data, 4))

        rospy.loginfo(rospy.get_caller_id() + "I heard %s i'll send %s",
data.data, data_send)

```



```

        # Sends the info over the UDP socket

        self.socket.sendto(data_send, self.address)

        logging.info(self.name + " callback " + "FINISH"+
str(time.time()))

class listenerExp(listenerGeneric):

    def listen(self):

        logging.info(self.name + " listen " + "START"+ str(time.time()))

        node_name = self.name + "_listener"

        rospy.init_node(node_name, anonymous=True)

        rospy.Subscriber("expansion", Float64MultiArray, self.callback)

        # spin() simply keeps python from exiting until this node is
stopped

        rospy.spin()

    def callback(self, data):

        logging.info(self.name + " callback " + "START"+
str(time.time()))

        ds_serialized = json.dumps(data.data)

        # rospy.loginfo(rospy.get_caller_id() + "I heard %s i'll send
%s", data.data, ds_serialized)

        self.socket.sendto(ds_serialized, self.address)

        logging.info(self.name + " callback " + "FINISH"+
str(time.time()))

```

```

# Class designed to read from the calibration publishers and transmit
# the data obtained through TCP it's sockets
class listenerCalibration(listenerGeneric):

    lock = threading.Lock()

    def listen(self):

        server_thread = threading.Thread(name=(self.name + "_server"),
target=self.server, args=[])

        server_thread.start()

        node_name = self.name + "_listener"

        rospy.init_node(node_name, anonymous=True)

        rospy.Subscriber(self.name, Float64MultiArray, self.callback)

        rospy.spin()

    def server(self):

        # file_name = self.name + '.txt'

        if self.name == "calibration_max":

            file_name = "calibration_max.txt"

        else:

            file_name = "calibration_min.txt"

        s = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)# Configure
sock_stream

```

```

sock_addr = address + self.name

try:
    os.unlink(sock_addr) # Remove (delete) the file path
except OSError:
    if os.path.exists(sock_addr):
        logging.warning("OSError: " + sock_addr)
        raise

s.bind(sock_addr)

s.listen(1)

while(True):
    connection, client_address = s.accept()

    # Get lock
    self.lock.acquire()

    try:
        with open(file_name, 'r') as input_cal:
            cal_array = array('f')
            cal_array.fromstring(input_cal.read())
            data_send = cal_array.tolist()

    finally:
        self.lock.release()

    ds_serialized = json.dumps(data_send)

    connection.sendall(ds_serialized.encode())

```

```

        connection.close()

def callback(self, data):
    file_name = self.name + '.txt'

    ds_deserialized = data.data

    self.lock.acquire()

    try:
        with open(file_name, 'wb') as write_cal:
            float_array = array('f', ds_deserialized)
            float_array.tofile(write_cal)

    finally:
        self.lock.release()

#####      Listeners' classes      #####

#####      Listeners' launchers      #####

def launchListenerFingers(name, id_p):
    setupLoggerFile('main_sub', 'INFO', 'a')
    logging.info("startListenerFingers START" + str(time.time()))
    s = socket.socket(socket.AF_UNIX, socket.SOCK_DGRAM)
    listener_obj = listenerFingers(name, id_p, s)
    listener_obj.run()
    logging.info("startListenerFingers FINISH" + str(time.time()))

```

```

def launchListenerExp(name, id_p):

    setupLoggerFile('main_sub', 'ERROR', 'a')

    logging.info("launchListenerExp START" + str(time.time()))

    s = socket.socket(socket.AF_UNIX, socket.SOCK_DGRAM)

    listener_obj = listenerExp("expansion", id_p, s)

    listener_obj.run()

    logging.info("launchListenerExp FINISH" + str(time.time()))

def launchListenerCalibration(name, id_p):

    setupLoggerFile('main_sub', 'ERROR', 'a')

    logging.info("launchListenerCalibration START" + str(time.time()))

    listener_obj = listenerCalibration(name, id_p)

    listener_obj.run()

    logging.info("launchListenerCalibration FINISH" + str(time.time()))

##### Listeners' launchers #####

##### Logger #####

def setupLoggerFile(name, level, mode):

    logger_file = './log/' + name + '_' + level + '_' +
str(time.strftime("%Y%m%d")) + '.log'

    print("Logger file: " + logger_file)

    if level is "DEBUG":

```

```
logging.basicConfig(format='% (asctime)s %(message)s ',
                    datefmt='%m/%d/%Y %I:%M:%S %p',
                    filename=logger_file,
                    filemode=mode,
                    level=logging.DEBUG)

elif level is "WARNING":

    logging.basicConfig(format='% (asctime)s %(message)s ',
                        datefmt='%m/%d/%Y %I:%M:%S %p',
                        filename=logger_file,
                        filemode=mode,
                        level=logging.WARNING)

else:

    logging.basicConfig(format='% (asctime)s %(message)s ',
                        datefmt='%m/%d/%Y %I:%M:%S %p',
                        filename=logger_file,
                        filemode=mode,
                        level=logging.INFO)
```

#####

Logger

#####

Anexo V – Código main_sub.py

```
#      main_sub.py
# Developed by Pablo San José
#      License CC-BY-SA      ***/

#!/usr/bin/python

from listenerGeneric import *

#####          Main          #####

if __name__ == '__main__':

    setupLoggerFile('main_sub', 'INFO', 'w')

    logging.info("Processes creation START" + str(time.time()))

    p=[]

    # Code to launch processes together

    p = [Process(target=launchListenerFingers, args=(nl_fingers[i], i))
for i in range(len(nl_fingers))]

    p.append(Process(target=launchListenerExp, args=("sep", 11))) #
Process for interdigital sensors
```



```

logging.info("Processes creation FINISH" + str(time.time()))

logging.info("Processes starting START" + str(time.time()))

# This must be separated in two different for loops
for process in p:
    process.start()

logging.info("Processes starting START" + str(time.time()))

logging.info("Processes joining START" + str(time.time()))

for process in p:
    process.join()

    logging.info("Processes joining FINISH" + str(time.time())) # Check
out this one

#####          Main          #####

```

Anexo VI – Código cal_sub.py

```
#      cal_sub.py
# Developed by Pablo San José
#      License CC-BY-SA      ***/

from listenerGeneric import *

#####          Main          #####

if __name__ == '__main__':

    setupLoggerFile('cal_sub', 'ERROR')

    # Spawning of the processes

    c=[]

        c.append(Process(target=launchListenerCalibration,
args=("calibration_max", 20)))

        c.append(Process(target=launchListenerCalibration,
args=("calibration_min", 21)))

    # Launching

    for cp in c:

        cp.start()
```

```
# Joining
for cp in c:
    cp.join()
```

```
#####          Main          #####
```

Anexo VII – Código blender_script.py

```
#     blender_script.py
# Developed by Pablo San José
#     License CC-BY-SA     ***/

import bge
import GameLogic as GL
import math
import json
import socket
import os
import sys
import threading
import time
import logging

from datetime import datetime

##### Thread #####

# Main function for moverZ_threads
# thread_bone -> object channel (bone)
def moverZ(thread_bone, owner, socket):
```

```

        GL.logging.info(threading.current_thread().name + "START: " +
str(time.time()))

        setupLoggerFile("INFO", "a")

data = -1

try:
    data, addr = socket.recvfrom(40)
except:
    pass

if data == -1:
    GL.logging.debug("No data yet!")
    # print("no data")
else:
    GL.logging.info("Data received: " + str(data))
    applyRotationZ(thread_bone, data)
    owner.update()

    GL.logging.info(threading.current_thread().name + "FINISH: " +
str(time.time()))

# Rotation function for moverZ_threads
def applyRotationZ(bone, data):
    rot_z = GL.lut1[float(data)]
    bone.rotation_mode = GL.ROT_MODE_ZYX

```

```

bone.rotation_euler=[0,0,rot_z] #BL_ArmatureChannel.rotation_euler

#print("Bone moved: ", bone.name)

def moverX(owner, socket):

    setupLoggerFile("INFO", "a")

    GL.logging.info(threading.current_thread().name + "START: " +
str(time.time()))

    data = ""

    try:

        data, addr = socket.recvfrom(40)

    except:

        pass

    # print with showing off purpose only

    if len(data) == 0:

        print("no data")

        GL.logging.debug("No data yet!")

    else:

        print("data -> ", data)

        data = str(data,'utf-8')

        GL.logging.info("Data received: " + data +
threading.current_thread().name + str(time.time()))

        applyRotationX(data)

        owner.update()

```

```
GL.logging.info(threading.current_thread().name + "FINISH: " +
str(time.time()))
```

```
# Rotation function for moverX_threads
```

```
def applyRotationX(data):
```

```
    data_ds = json.loads(data)
```

```
    print("data_ds -> ", data_ds)
```

```
    for bone in GL.x_bones:
```

```
        bone.rotation_mode = GL.ROT_MODE_ZYX
```

```
    rot_thumb = GL.lut2[round(data_ds[0], 4)] # Pulgar
```

```
        GL.x_bones[4].rotation_euler=[rot_thumb,      0,      0]
#BL_ArmatureChannel.rotation_euler
```

```
    v1 = GL.lut2[round(data_ds[1], 4)]
```

```
    v2 = GL.lut2[round(data_ds[2], 4)]
```

```
    rot4 = GL.lut2[round(data_ds[3], 4)]
```

```
    rot1 = v1 - v2
```

```
    if rot1 > 0:
```

```
        rot2 = v1 - rot1
```

```
        rot3 = v2
```

```
        GL.x_bones[0].rotation_euler = [rot2/4, 0, 0]
#BL_ArmatureChannel.rotation_euler
```

```
        GL.x_bones[1].rotation_euler = [rot1/4, 0, 0]
```

```

#BL_ArmatureChannel.rotation_euler

        GL.x_bones[2].rotation_euler = [(rot1 + rot3)/4, 0, 0]
#BL_ArmatureChannel.rotation_euler

        GL.x_bones[3].rotation_euler = [(rot1 + rot3 + rot4)/4, 0, 0]
#BL_ArmatureChannel.rotation_euler

else:

    rot2 = v2-rot1

    rot3 = v1

        GL.x_bones[0].rotation_euler = [(rot1 + rot3)/4, 0, 0]
#BL_ArmatureChannel.rotation_euler

        GL.x_bones[1].rotation_euler = [rot1/4, 0, 0]
#BL_ArmatureChannel.rotation_euler

        GL.x_bones[2].rotation_euler = [rot2/4, 0, 0]
#BL_ArmatureChannel.rotation_euler

        GL.x_bones[3].rotation_euler = [(rot2 + rot4)/4, 0, 0]
#BL_ArmatureChannel.rotation_euler

##### End Thread #####

##### Global Variables #####

nl_fingers = ["indice1", "indice2",

              "corazon1", "corazon2",

              "anular1", "anular2",

              "menique1", "menique2",

              "pulgar1", "pulgar2"]

nl_sep = ["pulgارينdice", "indicecorazon",

          "corazonanular", "anularmenique"]

```



```

not_allowed = ["hand.wrist", "index.d_phalanx", "middle.d_phalanx",
"ring.d_phalanx", "pinky.d_phalanx", "thumb.methacarpal"]

x_bones = ["index.p_phalanx", "middle.p_phalanx", "ring.p_phalanx",
"pinky.p_phalanx", "thumb.methacarpal"]

num_sockets = 11

##### End Global Variables #####

##### StartUp Functions #####

def setupLoggerFile(level, mode):

    logger_file = './log/udp_blenthread_' + level + '_' +
str(time.strftime("%Y%m%d")) + '.log'

    if level is "DEBUG":

        logging.basicConfig(format='%(asctime)s %(message)s',

            datefmt='%m/%d/%Y %I:%M:%S %p',

            filename=logger_file,

            filemode=mode,

            level=logging.DEBUG)

    elif level is "WARNING":

        logging.basicConfig(format='%(asctime)s %(message)s',

            datefmt='%m/%d/%Y %I:%M:%S %p',

            filename=logger_file,

            filemode=mode,

            level=logging.WARNING)

    else:

```

```
logging.basicConfig(format='%(asctime)s %(message)s',
                    datefmt='%m/%d/%Y %I:%M:%S %p',
                    filename=logger_file,
                    filemode=mode,
                    level=logging.INFO)
```

```
GL.logging = logging
```

```
def setupBonesLists(hand):
```

```
    bones_list = [bone for bone in hand.channels if bone.name not in
not_allowed]
```

```
    GL.num_bones = len(bones_list)
```

```
    x_bones = [bone for bone in hand.channels if bone.name in x_bones]
```

```
    GL.bones_list = bones_list
```

```
    GL.x_bones = x_bones
```

```
def calibration():
```

```
    address = '/tmp/'
```

```
    addr_cal = []
```

```
    addr_cal.append(address + 'calibration_max')
```

```
    addr_cal.append(address + 'calibration_min')
```

```
    sock_cal = []
```

```

sock_cal.append(socket.socket(socket.AF_UNIX, socket.SOCK_STREAM))
sock_cal.append(socket.socket(socket.AF_UNIX, socket.SOCK_STREAM))

data_max = []

try:
    sock_cal[0].connect(addr_cal[0])
    sock_cal[1].connect(addr_cal[1])
except socket.error:
    GL.logging.error("Error when connecting to calibration sockets" +
str(time.time()))
    raise

try:
    data = sock_cal[0].recv(168).decode()
    logging.info("Calibration max: " + data)
    data_max = json.loads(data)
    data = sock_cal[1].recv(168).decode()
    data_min = json.loads(data)
    logging.info("Calibration max: " + data)
except:
    GL.logging.error("Error when reading from calibration sockets" +
str(time.time()))
    raise

for sock in sock_cal:
    sock.close()

```

```

def setupSockets():
    address = '/tmp/'

    # Sockets setup

    s = [socket.socket(socket.AF_UNIX, socket.SOCK_DGRAM) for i in
range(num_sockets)]

    socket_address = [address + nl_fingers[i] for i in range(num_sockets-
1)]

    socket_address.append(address + "expansion")

    for s_address in socket_address:

        # Make sure the socket does not already exist

        try:

            os.unlink(s_address) # Remove (delete) the file path

        except OSError:

            if os.path.exists(s_address):

                GL.logging.warning("OSError: " + s_address)

                raise

    for i in range(num_sockets):

        s[i].setblocking(0)

        s[i].bind(socket_address[i])

GL.s=s

```

```

def fill_luts():

    lut1 = {x/10000 : round(-1*x/10000 * math.radians(90), 4) for x in
range (0, 10001)}

    GL.lut1 = lut1

    lut2 = {x/10000 : round(1*x/10000 * math.radians(45), 4) for x in
range (0, 10001)}

    GL.lut2 = lut2

##### End StartUp Functions #####

if __name__ == '__main__':

    # Get the controller of the scene, so we can manipulate it
    cont = bge.logic.getCurrentController()

    hand = cont.owner

    # Block of code executed only once
    if not hand['StartUp']:

        # Setup logger file
        setupLoggerFile("INFO", "w") #DEBUG - INFO - WARNING

        GL.logging.debug("StartUp setup")

        GL.logging.info("Only once block START: " + str(time.time()))

```

```

#Setup bones lists

try:
    setupBonesLists(hand)
except:
    GL.logging.error("Error extracting bones' names from the
scene")

# Calibration (Optional)

try:
    calibration()
except:
    GL.logging.error("Error receiving calibration")

#Filling of the LookUp Tables

try:
    fill_luts()
except:
    GL.logging.error("Error filling LUT")

# Setup of all the sockets

try:
    setupSockets()
except:
    GL.logging.error("Error creating sockets")

hand['StartUp'] = True

GL.logging.info("Only once block FINISH: " + str(time.time()))

```

```
else:
```

```
    GL.logging.info("Threads' creation START: " + str(time.time()))
```

```
    # Creation of the threads (ALL OK)
```

```
    try:
```

```
        mover_threads = [threading.Thread(name=GL.bones_list[i],  
target=moverZ, args=[GL.bones_list[i], hand, GL.s[i]]) for i in  
range(GL.num_bones)]
```

```
        mover_threads.append(threading.Thread(name="btwFingers",  
target=moverX, args=[hand, GL.s[-1]]))
```

```
    except:
```

```
        GL.logging.error("Error creating threads")
```

```
for t in mover_threads:
```

```
    try:
```

```
        t.start()
```

```
    except:
```

```
        GL.logging.error("Error starting threads")
```

```
for t in mover_threads:
```

```
    try:
```

```
        t.join()
```

```
    except:
```

```
        GL.logging.error("Error joining threads for Z")
```

```
GL.logging.info("Threads' creation FINISH: " + str(time.time()))
```

Anexo VIII – Código gui_script.py

```
#      gui_script.py
# Developed by Pablo San José
#      License CC-BY-SA      ***/

#!/usr/bin/python3

import tkinter as tk
import tkinter.messagebox
from PIL import ImageTk, Image
from subprocess import call as command
import os
import signal
from multiprocessing import Process

##### Constants #####

logging_levels=["DEBUG", "INFO", "WARNING"]
logging_value=2

##### End Constants #####
```



```

##### Callbacks #####

def gloveCallBack():
    tk.messagebox.showinfo( "Attention!", "Launching\nGlove Driver")
    launcher = Process(target=gloveLauncher, args=())
    launcher.start()

def manoCallBack():
    tk.messagebox.showinfo( "Attention!", "Launching\nROS Publishers")
    launcher = Process(target=manoLauncher, args=())
    launcher.start()

def subscribersCallBack():
    tk.messagebox.showinfo( "Attention!", "Launching\nROS Listeners")
    launcher = Process(target=subsLauncher, args=())
    launcher.start()

def calibrationCallBack():
    tk.messagebox.showinfo( "Attention!", "Launching\nROS Listeners")
    launcher = Process(target=calSubsLauncher, args=())
    launcher.start()

def blenderCallBack():
    tk.messagebox.showinfo( "Attention!", "Launching\nBlender")
    launcher = Process(target=blenderLauncher, args=())
    launcher.start()

```

```

def roscoreCallBack():
    tk.messagebox.showinfo( "Attention!", "Launching\nROSCore")
    launcher = Process(target=roscoreLauncher, args=())
    launcher.start()

def compilationCallBack():
    tk.messagebox.showinfo( "Compilation", "Compilation\nunder way")
    launcher = Process(target=compilationLauncher, args=())
    launcher.start()

##### End Callbacks #####

##### Launchers #####

def gloveLauncher():
    command(["xterm", "-e", "sudo", "../ ../ ../devel/lib/tfg/dglove"])

def manoLauncher():
    command(["xterm", "-e", "../ ../ ../devel/lib/tfg/mano"])

def calSubsLauncher():
    command(["xterm", "-e", "python", "./cal_sub.py"])

def subsLauncher():
    command(["xterm", "-e", "python", "./main_sub.py"])

```

```

def blenderLauncher():
    command(["xterm", "-e", "blender", "./dgHand.blend"])

def roscoreLauncher():
    command(["xterm", "-e", "roscore"])

def compilationLauncher():
    command(["xterm", "-e", "./compilation.sh"])

##### End Launchers #####

def finish():
    os.killpg(os.getpid(), signal.SIGTERM)

root = tkinter.Tk()

root.wm_title("Interfaz Gráfico para 5DT Data Glove")

# R0 C0-4

title_image = ImageTk.PhotoImage(Image.open("./img/titulo2.png"))
title = tk.Label(root, image=title_image)
title.grid(row=0, columnspan=5, sticky='W'+ 'E'+ 'N'+ 'S', padx=5, pady=5)

# R1-2 C0

glove_launcher = tk.Button(text="Launch\nGlove", command=gloveCallBack)

```

```

glove_launcher.grid(row=1, rowspan=2, column=0, sticky='W'+ 'E'+ 'N'+ 'S',
padx=5, pady=5)

# R1-2 C1

mano_launcher = tk.Button(text="Launch\nPublishers",
command=manoCallBack)

mano_launcher.grid(row=1, rowspan=2, column=1, sticky='W'+ 'E'+ 'N'+ 'S',
padx=5, pady=5)

# R1-2 C2

ros_icon = ImageTk.PhotoImage(Image.open("./img/ros_icon.png"))

ros_label = tk.Label(root, image=ros_icon)

ros_label.grid(row=1, rowspan=2, column=2, sticky='W'+ 'E'+ 'N'+ 'S',
padx=5, pady=5)

# R1 C3

subscribers_launcher = tk.Button(text="Launch\nSubscribers",
command=subscribersCallBack)

subscribers_launcher.grid(row=1, rowspan=1, column=3,
sticky='W'+ 'E'+ 'N'+ 'S', padx=5, pady=5)

# R2 C3

calibration_launcher = tk.Button(text="Launch\nCal Subs",
command=calibrationCallBack)

calibration_launcher.grid(row=2, rowspan=1, column=3,
sticky='W'+ 'E'+ 'N'+ 'S', padx=5, pady=5)

# R1 C4

blender_launcher = tk.Button(text="Launch\nBlender",
command=blenderCallBack)

blender_launcher.grid(row=1, rowspan=2, column=4, sticky='W'+ 'E'+ 'N'+ 'S',
padx=5, pady=5)

```

```

# R3 C2

roscore_launcher = tk.Button(text="Launch\nROSCore",
command=roscoreCallBack)

roscore_launcher.grid(row=3, rowspan=1, column=2, sticky='N', padx=5,
pady=0)

# R3 C0

compilation_launcher = tk.Button(text="Compile",
command=compilationCallBack)

compilation_launcher.grid(row=3, rowspan=1, columnspan=2, column=0,
sticky='N' + 'S' + 'E' + 'W', padx=5, pady=0)

# LAST

quit = tk.Button(text="QUIT", fg="red", command=finish)

quit.grid(row=8, column=4, padx=1, pady=1)

root.mainloop()

##### End script #####

```

Anexo IX – Código script compilation.sh

```
#!/bin/bash
```

```
g++ ../src/dglove.cpp -lfglove -o ../../../../devel/lib/tfg/dglove
```

```
cd ../../../../ && catkin_make
```

```
sleep 3
```


Anexo X – Código CmakeLists.txt

```
cmake_minimum_required(VERSION 2.8.3)

project(tfg)

## Find catkin macros and libraries

## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)

## is used, also find other catkin packages

find_package(catkin REQUIRED COMPONENTS

    roscpp

    rospy

    std_msgs

)
```



```
#####
```

```
## Build ##
```

```
#####
```

```
## Specify additional locations of header files
```

```
## Your package locations should be listed before other locations
```

```
include_directories(
```

```
    ${catkin_INCLUDE_DIRS}
```

```
)
```

```
## Declare a C++ executable
```

```
add_executable(mano src/mano.cpp)
```

```
## Add cmake target dependencies of the executable
```

```
## same as for the library above
```

```
add_dependencies(mano mano_generate_messages_cpp)
```

```
# Specify libraries to link a library or executable target against
```

```
target_link_libraries(mano ${catkin_LIBRARIES})
```


Anexo XI – Código packet.xml

```
<?xml version="1.0"?>
<package>
  <name>tfg</name>
  <version>1.0.0</version>
  <description>My TFG Package</description>

  <!-- One maintainer tag required, multiple allowed, one person per tag
-->

  <!-- Example: -->

  <!-- <maintainer email="jane.doe@example.com">Jane Doe</maintainer> -->
          <maintainer          email="pabloguinness@gmail.com">email
maintainer</maintainer>

  <!-- One license tag required, multiple allowed, one license per tag
-->

  <license>CC-BY-SA</license>

  <!-- Author tags are optional, mutiple are allowed, one per tag -->
  <!-- Authors do not have to be maintianers, but could be -->
  <!-- Example: -->
  <author email="pabloguinness@gmail.com">Pablo Sanjo</author> -->
```

```
<!-- The *_depend tags are used to specify dependencies -->
<!-- Dependencies can be catkin packages or system dependencies -->
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
<run_depend>roscpp</run_depend>
<run_depend>rospy</run_depend>
<run_depend>std_msgs</run_depend>

</package>
```