UNIVERSIDADE DA CORUÑA

Escola Politécnica Superior

Trabajo Fin de Grado
CURSO 2016/17

Sistema de comunicación con dispositivos hápticos TOUCH 3D

Grado/Máster en Ingeniería Mecánica

ALUMNA/O

Aitor García Catoira

TUTORAS/ES

Manuel Jesús González Castro

Alberto Luaces Fernández

FECHA

JULIO 2017

Resumen del proyecto

El presente proyecto consiste en el estudio de la incorporación de dispositivos hápticos a un entorno de ensamblaje industrial. Esta metodología permite un aprendizaje más seguro y económico de las tareas de montaje de un producto por parte de los diseñadores de los sistemas, o de los operarios encargados de tales labores.

Actualmente la mayoría de los programas CAD sólo permiten la visualización simplificada de las tareas necesarias para el montaje de un producto; gracias al empleo de dispositivos hápticos se puede incluir el sentido del tacto. Además, hace posible detectar problemas en el proceso de ensamblaje, que reflejan situaciones imposibles que sólo se apreciarían en la realidad.

Los dispositivos hápticos suelen ser empleados para dos tareas principales: la captura del movimiento y la representación de fuerzas acordes al estado de una simulación.

Los hápticos se han ido introduciendo cada vez más en entornos como la medicina, artes gráficas o entretenimiento, pero menos en la industria.

El objetivo del proyecto es el desarrollo de un simulador de ensamblaje virtual con retroalimentación de fuerzas que permita detectar la sencillez y la factibilidad de un plan de montaje determinado, compuesto a partir de piezas diseñadas en CAD.

El estudio concluye con una demostración de la implementación de una metodología de ensamblaje colaborativo, en la cual pueden participar varias personas simultáneamente para realizar las tareas necesarias.

Resumo do proxecto

Este proxecto consiste no estudo da incorporación de dispositivos hápticos a un entorno de ensamblaxe industrial. Esta metodoloxía permite unha aprendizaxe máis segura e económica das tarefas de montaxe dun produto por parte dos deseñadores dos sistemas, ou dos operarios responsables de tales tarefas.

Actualmente a maioría dos programas CAD só permiten a visualización simplificada das tarefas necesarias para a montaxe dun produto; grazas ó empleo de dispositivos hápticos pódese incluír o sentido do tacto. Tamén fai que sexa posible detectar problemas no proceso de ensamblaxe, que reflexan situacións imposibles apreciables só na realidade.

Os dispositivos hápticos adoitan ser empregados para dúas tarefas principais: a captura do movemento e a representación de forzas xeradas acordes co estado dunha simulación.

Os hápticos están introducidos cada vez máis en ambientes como, a medicina, artes gráficas ou entretemento, pero menos na industria.

O obxectivo do proxecto é desenvolver un simulador de ensamblaxe virtual con retroalimentación de forzas para detectar a sinxeleza e a viabilidade dun plan de montaxe determinado, composto por pezas deseñadas en CAD.

O estudo conclúe cunha demostración da implantación dunha metodoloxía de ensamblaxe colaborativo, no que varias persoas poden participar simultaneamente para a realización das tarefas necesarias.

Project Summary

The present project is about the study of the incorporation of haptics devices to an environment of industrial assembly. This methodology allows a surer and economic learning of the tasks of assembly of a product on the part of the designers of the systems, or of the operatives in charge of such labors.

Actually the majority of the CAD programs only allow the visualization simplified of the tasks necessary for the assembly of a product; thanks to use of haptics devices it is possible to include a sense to major: the tact. In addition, it makes possible to detect problems in the process of assembly which reflect impossible situations that would only be appreciated in reality.

The haptics devices are often used for two principal tasks: to catch the movement and to represent identical forces to the condition of a simulation.

The haptics have been introduced in environments as the medicine, graphical arts or the entertainment, but in the industry not much.

The aim of this project is the development of a malingerer of virtual assembly with feedback of forces that allows detecting the simplicity and the feasibility of a plan of certain assembly, composed from pieces designed in CAD's environment.

The study concludes with a demonstration of the implementation of a methodology of collaborative assembly, in which several persons can take part simultaneously to realize the necessary tasks.

UNIVERSIDADE DA CORUÑA

Escola Politécnica Superior

TRABAJO FIN DE GRADO/MÁSTER
CURSO 2016/17

Sistema de comunicación con dispositivos hápticos TOUCH 3D

Grado/Máster en Ingeniería Mecánica

Documento

MEMORIA

INDICE

INDICE DE FIGURAS

1. INTRODUCCIÓN

La motivación de realizar este proyecto ha sido es el estudio de la incorporación de dispositivos hápticos a un entorno de ensamblaje industrial. Esta implementación en los softwares de CAD permitiría un aprendizaje más seguro y económico de las tareas de montaje de un producto hecho de subcomponenetes, por parte de los diseñadores de los sistemas, o de los operarios encargados de tales labores.

Debido a que actualmente la mayoría de los programas CAD sólo permiten la visualización simplificada de las tareas necesarias para el montaje de un producto; gracias al empleo de dispositivos hápticos se puede incluir un sentido a mayores: el sentido del tacto. Además, hace posible detectar problemas en el proceso de ensamblaje, que reflejan situaciones imposibles que sólo se apreciarían en la realidad.

Los dispositivos hápticos suelen ser empleados para dos tareas principales: la captura del movimiento y la representación de fuerzas acordes al estado de una simulación.

Los hápticos se han ido introduciendo cada vez más en entornos como la medicina, artes gráficas o entretenimiento, pero menos en la industria.

El objetivo del proyecto es el desarrollo de un simulador de ensamblaje virtual con retroalimentación de fuerzas que permita detectar la sencillez y la factibilidad de un plan de montaje determinado, compuesto a partir de piezas diseñadas en CAD. El caso que se propone es el montaje de un ensamblaje que consistiría en levantar dos caballetes del suelo, abrirlos y colocar una pieza de chapa metálica encima de estos.

El estudio concluye con una demostración de la implementación de una metodología de ensamblaje colaborativo, en la cual pueden participar varias personas simultáneamente para realizar las tareas necesarias.

Para comenzar se profundizará en que consiste un ensamblaje virtual y los dispositivos hápticos explicando su funcionamiento.

1.1 Ensamblaje virtual

Entendemos por ensamblaje virtual todo montaje de un conjunto de piezas en un ambiente virtual que simule la realidad, el cual ayude a la visualización del proceso en un ambiente computacional.

Hoy en día existen algunos softwares de CAD que permiten el ensamblaje virtual de un producto, por ejemplo, SolidEdge, SolidWorks, NX, AutoDesk…

Los softwares de CAD o El diseño asistido por ordenador, más conocido por sus siglas inglesas CAD (computer-aided design), es el uso de un amplio rango de herramientas computacionales que asisten a ingenieros, arquitectos y diseñadores. El CAD es también utilizado en el marco de procesos de administración del ciclo de vida de productos.

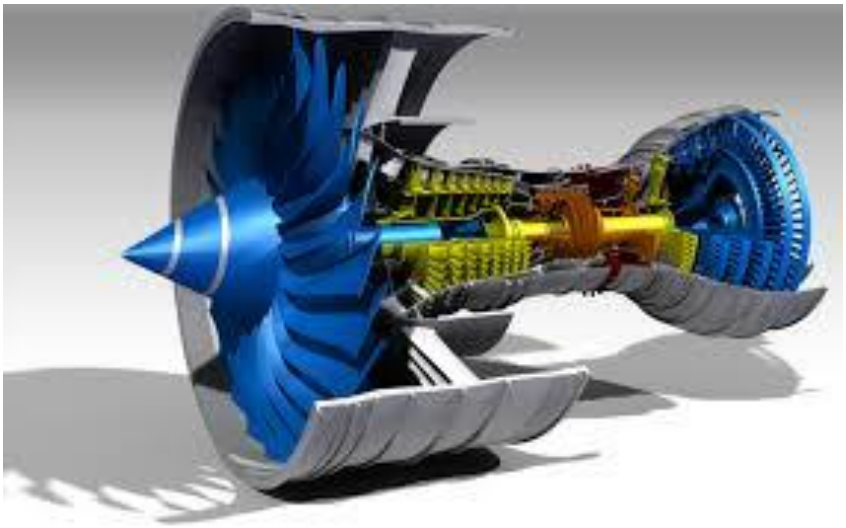


Figura 1. Ensamblaje con CAD

Estas herramientas se pueden dividir básicamente en programas de dibujo 2D y de modelado 3D. Las herramientas de dibujo en 2D se basan en entidades geométricas vectoriales como puntos, líneas, arcos y polígonos, con las que se puede operar a través de una interfaz gráfica. Los modeladores en 3D añaden superficies y sólidos.

El usuario puede asociar a cada entidad una serie de propiedades como color, capa, estilo de línea, nombre, definición geométrica, material, etc., que permiten manejar la información de forma lógica.

Además se pueden renderizar a través de diferentes motores o softwares como V-Ray, Maxwell Render, Lumion, Flamingo, entre los que son pagos, hay algunos de licencia free and open source como por ejemplo el Kerkythea y Aqsis, entre los más usados, son modeladores 3D para obtener una previsualización realista del producto, aunque a menudo se prefiere exportar los modelos a programas especializados en visualización y animación, como Autodesk Maya, Bentley MicroStation, Softimage XSI o Cinema 4D y la alternativa libre y gratuita Blender, capaz de modelar, animar y realizar videojuegos. Blender fue el usado en este estudio.

El CAD es una parte de toda la actividad de desarrollo de productos digitales dentro de los procesos PLM (Product Lifecycle Management), y como tal se utiliza junto con otras herramientas, ya sean módulos integrados o productos independientes, como, por ejemplo:

- Ingeniería Asistida por Ordenador (CAE) y Análisis de Elementos Finitos (FEA).
- Fabricación asistida por computadora (CAM) incluyendo instrucciones a las máquinas del control numérico del ordenador (CNC).
- Representación fotorrealista y simulación de movimiento.
- Gestión de documentos y control de revisiones mediante la gestión de datos de producto (PDM).

CAD ha demostrado ser útil para los ingenieros utilizando cuatro propiedades que son historial, características, parametrización y restricciones de alto nivel. La historia de la

construcción se puede utilizar para mirar hacia atrás en las características personales del modelo y trabajar en el área única en lugar de todo el modelo. Parámetros y restricciones pueden usarse para determinar el tamaño, la forma y otras propiedades de los diferentes elementos de modelado.

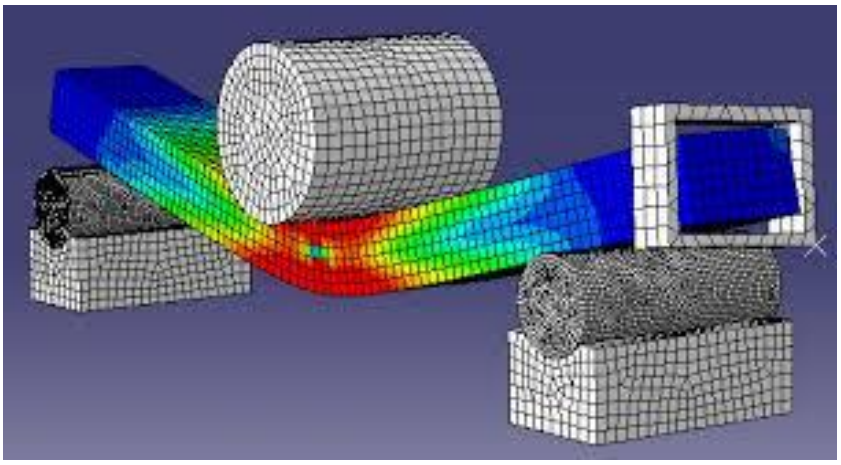


Figura 2. Análisis mediante elementos finitos con ayuda de CAD

Las características del sistema CAD pueden utilizarse para la variedad de herramientas de medida tales como la resistencia a la tracción, el límite de elasticidad, las propiedades eléctricas o electromagnéticas. También su estrés, tensión, tiempo o cómo el elemento se ve afectado en ciertas temperaturas, etc.

Hay dos tipos de Modelado Sólido 3D:

- El modelado paramétrico permite al operador utilizar lo que se conoce como "intención de diseño". Los objetos y características creados son modificables. Cualquier modificación futura puede hacerse cambiando cómo se creó la parte original. Si se pretende que una característica esté ubicada desde el centro de la pieza, el operador debe localizarla desde el centro del modelo. La característica podría localizarse utilizando cualquier objeto geométrico ya disponible en la pieza, pero esta colocación aleatoria anularía la intención del diseño. Si el operador diseña la pieza a medida que funciona, el modelador paramétrico puede realizar cambios en la parte manteniendo relaciones geométricas y funcionales.

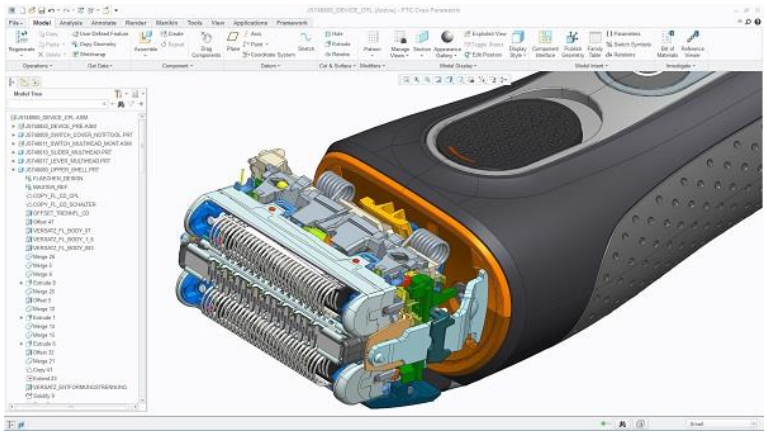


Figura 3. Ejemplo modelado paramétrico

- El modelado directo o explícito proporciona la posibilidad de editar geometría sin un árbol de historial. Con el modelado directo una vez que se utiliza un boceto para crear geometría, el boceto se incorpora a la nueva geometría y el diseñador simplemente modifica la geometría sin necesidad del boceto original. Al igual que con el modelado paramétrico, el modelado directo tiene la capacidad de incluir relaciones entre geometría seleccionada (por ejemplo, tangencia, concentricidad).

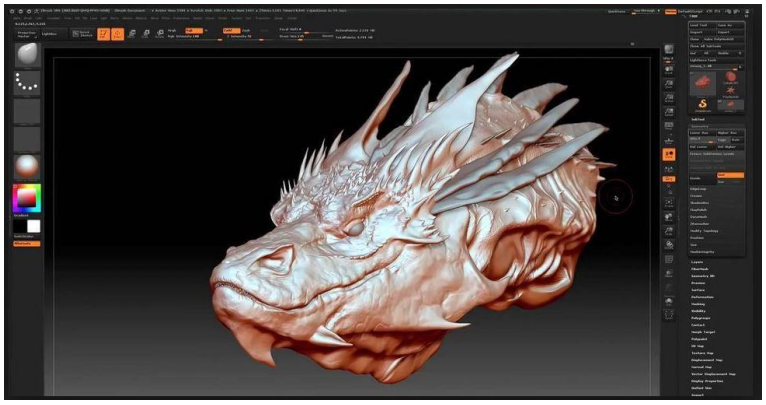


Figura 4. Ejemplo modelado directo

Sin embargo, de todas estas ventajas y aplicaciones, no se ha tenido en cuenta llevar la realidad del montaje a un siguiente nivel: dar un sentido a mayores (el tacto) que complementaría la tarea de montaje, ayudando al usuario final con un adiestramiento y la verificación de factibilidad y sencillez. Esto sería posible mediante la implementación de dispositivos hápticos que complementen estos softwares.

1.2 DISPOSITIVOS DE REPRESENTACIÓN DE FUERZAS: HÁPTICOS

Para interactuar con un dispositivo, las personas pueden utilizar diferentes canales sensoriales: visual, auditivo y háptico (relativo al sentido del tacto).

Puesto que el tema de interés son los dispositivos hápticos se explicará en qué consisten estos dispositivos, cómo es su interacción hombre-máquina, cuáles son sus ventajas, sus desventajas y dar ejemplos de dispositivos hápticos comerciales.

Con el término “interface háptico” aludimos a aquellos dispositivos que permiten al usuario tocar, sentir o manipular objetos simulados en entornos virtuales y sistemas tele-operados. Es decir, los dispositivos hápticos proporcionan la realimentación de fuerza (producción mecánica de información sensorial por el sistema kinésico, es decir por la sensación del movimiento, sensaciones originadas en el músculo, tendones y uniones) al sujeto que interactúa con entornos virtuales o remotos.

Tales dispositivos trasladan una sensación de presencia al operador. El usuario no sólo envía la información al ordenador, sino que también puede recibirla información del ordenador en forma de sensación sobre alguna parte del cuerpo.

Figura 5. Ejemplo de aplicación de los dispositivos hápticos

Estos dispositivos se pueden utilizar en varios paradigmas de interacción, pero con el que más encaja es con el de la realidad virtual. Estos dispositivos permiten un control sobre la realidad virtual más natural, permitiendo al usuario tocar y sentir esta realidad virtual.

En la mayoría de simulaciones realizadas en entornos virtuales, basta con emplear dispositivos de representación 3D y dispositivos de sonido 3D estéreo para provocar en el usuario, mediante imágenes y sonidos, la sensación de inmersión dentro del espacio virtual.

No obstante, además de provocar en el usuario esta sensación de inmersión, debemos proporcionarle la posibilidad de interactuar con el medio virtual, pudiendo establecer entre el usuario y el entorno virtual una transferencia bidireccional y en tiempo real de información mediante el empleo de interfaces de tipo háptico.

Algunas de las ventajas de la utilización de los dispositivos hápticos son la manipulación de los entornos virtuales de forma natural y en 3D no con proyecciones 2D, añadimos información multisensorial (lo cual facilita la realización de distintas tareas, mejora la inmersión y plausibilidad).

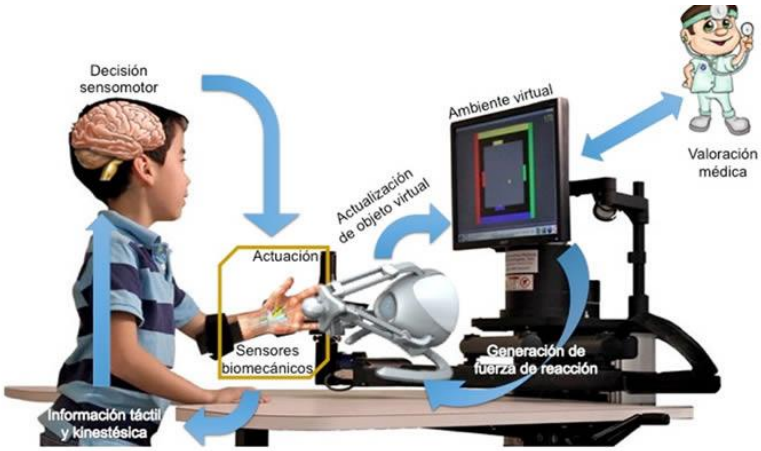


Figura 6. Usuario sin alto grado de conocimiento utilizando dispositivos hápticos

Además, se pueden integrar perfectamente con la mayoría de los dispositivos informáticos actuales, ya que estos sólo intentan introducir el sentido del tacto para interaccionar con la máquina y no intentan sustituir otros dispositivos actuales que normalmente utilizan otros sentidos diferentes al tacto para la interacción hombre-máquina.

Por otra parte, algunas de las desventajas de su utilización (aunque en el futuro puede que cambie su situación) son su alto coste, su complejidad a la hora de diseñar o fabricar estos dispositivos, la ausencia de estándares, reglas, leyes que permitan un desarrollo eficaz de estos dispositivos, la poca familiarización de la gran mayoría de usuarios con estos dispositivos...

Algunos de los principales campos de aplicación de los interfaces hápticos son:

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

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

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

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

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

Los dispositivos hápticos se han ido introduciendo cada vez más en entornos como la medicina, artes gráficas o el entretenimiento, pero no hay mucho uso en la industria.

La importancia de los interfaces hápticos es determinante en la realización de tareas en las que se requiera un alto grado de entrenamiento, como pueden ser: ensamblaje de conjuntos complejos antes de ser fabricados, etc. Ayudan a su vez, a incrementar la sensación de presencia o inmersión del usuario dentro de un entorno simulado, proporcionando restricciones naturales al movimiento de objetos.

Existen dos grandes familias de dispositivos hápticos según sus tipos de entradas y salidas:

a) Impedancia: el usuario mueve el dispositivo, y el dispositivo reaccionará con una fuerza si es necesario. Sus entradas son desplazamientos y sus salidas fuerzas.

b) Admitancia: el dispositivo mide las fuerzas que el usuario ejerce sobre él y reacciona con el movimiento (aceleración, velocidad, posición...). Sus entradas son fuerzas y sus salidas desplazamientos.

Los de impedancia han sido los empleados en este estudio y además son los más comunes, por lo que nos centraremos principalmente en los de esta clase. Los eventos que llevan a cabo en su flujo de control son:

1. Los sensores de desplazamiento recogen el movimiento.
2. Envío mediante driver de este desplazamiento al simulador software.
3. Interacción con el mundo virtual y cálculo de fuerzas.
4. Se envía señal al driver.
5. El dispositivo produce la fuerza necesaria.

También se pueden clasificar según:

a) Sus grados de libertad: 2DOF, 4DOF, 6DOF, etc.

b) El efector: tipo guante, tipo lápiz (fue el usado en este estudio).

c) Su ámbito de aplicación: específicos (medicina, por ejemplo), para propósito general.

d) El tipo de receptor sensorial: retroalimentación de fuerza, retroalimentación táctil, vestibulares, mixtos.

A continuación, se mencionarán algunos ejemplos de dispositivos hápticos comerciales:

- Guantes con feedback de fuerza CYBERGRASP Immersion Co: El Cybergrasp consiste en una estructura exoesquelética fijada a la parte posterior de la mano, que es accionada por unos actuadores instalados fuera de ésta, en una caja de control, con el objetivo de facilitar su manejo aligerando su peso, de aproximadamente 450 gr. La fuerza máxima que puede aplicar sobre cada dedo es de 12N.

Figura 7. Guantes Cybergrasp

- Guantes con feedback táctil CYBERTOUCH Immersion Co: Estos guantes son mucho más ligeros que los que poseen force feedback y emplean normalmente vibradores electromecánicos para proporcionar datos de texturas o rugosidades. El Cybertouch de Immersion Co. pesa solamente 144gr. Usa 6 vibradores electromecánicos situados en la parte posterior de los dedos y en la palma de la mano. Estos actuadores producen vibraciones de 0-125Hz, alcanzando unos 1.2N de fuerza a 125Hz.

- PHANTOM Omni® Haptic Device: Es un dispositivo con el cual, por ejemplo, los ciegos pueden aprender a escribir con seguridad ayudados por el software del profesor Stephen Brewster. Este software permite que un usuario escriba un texto y mediante el Phantom transmitir al invidente dicha escritura. El Phantom Omni es el dispositivo de tecnología haptica más económico del mercado.

Figura 8.Phantom Omni

- Touch 3D Stylus: El Touch 3D Stylus de 3DSystems es el posterior modelo al Omni, es uno de los dispositivos hápticos más económicos. Y fue precisamente

este, el que se ha empleado en este estudio. A continuación, se muestran sus especificaciones para así poder tener una orientación de que rangos de aplicación se ha podido tratar en la investigación.

	Touch 3D Stylus
Workspace	10.45 W x 9.5 H x 3.5 D in
Range of motion	Hand movement pivoting at wrist
Nominal position resolution	Approx 0.084 mm
Maximum exertable force and torque at nominal position (orthogonal arms)	3.4 N
Stiffness	-
Force feedback (6 Degrees of Freedom)	x, y, z
Position sensing/input (6 Degrees of Freedom)	x, y, z (digital encoders)
[Stylus gimbal]	[Roll, pitch, yaw (± 5% linearity potentiometers)]
Interface	USB 2.0

Figura 9. Especificaciones del dispositivo háptico empleado para este estudio

A continuación, se profundizará más en cómo es el funcionamiento de un dispositivo háptico y cuáles son los puntos importantes a tratar en su diseño.

1.3 FUNCIONAMIENTO DE UN DISPOSITIVO HÁPTICO

Los dispositivos hápticos de pueden medir de forma precisa la posición espacial 3D (a lo largo de los ejes X, Y y Z) y la orientación (giro, inclinación y dirección) del lápiz de mano.

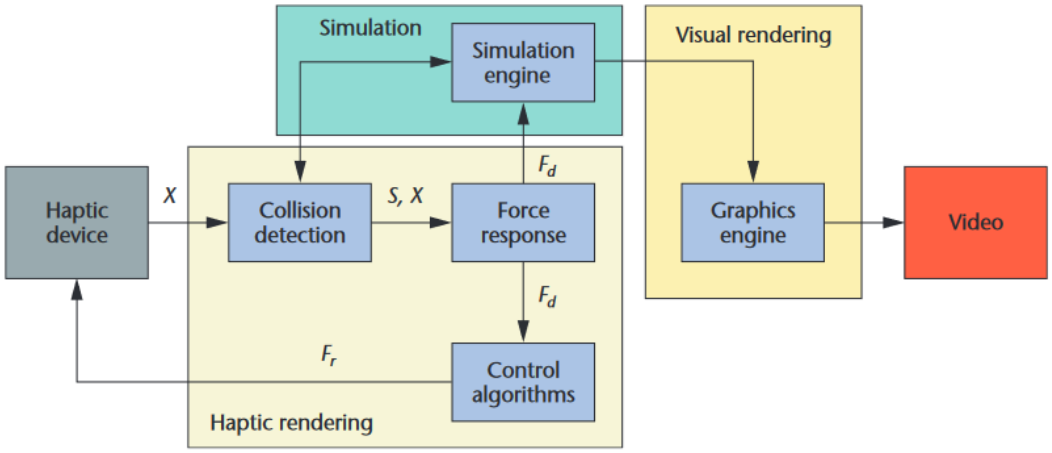


Figura 10. Flujo de control de un programa para simulación háptica

Los dispositivos utilizan una serie servomotores ubicados en el brazo para crear las fuerzas de retorno en la mano del usuario para simular el tacto y la interacción con objetos virtuales. Los dispositivos proporcionan una retroalimentación de fuerza de 3 o 6 grados de libertad (DOF), según puedan controlar sólo las posiciones del lápiz o también las rotaciones de este. Por ejemplo, el dispositivo que se empleó en este estudio tiene 3, es decir, sólo puede devolver una respuesta ante las traslaciones del puntero.

Los dispositivos hápticos utilizan servomotores para crear las fuerzas de retorno en la mano del usuario a fin de simular el tacto cuando el cursor interactúa con el modelo en 3D en el espacio virtual.

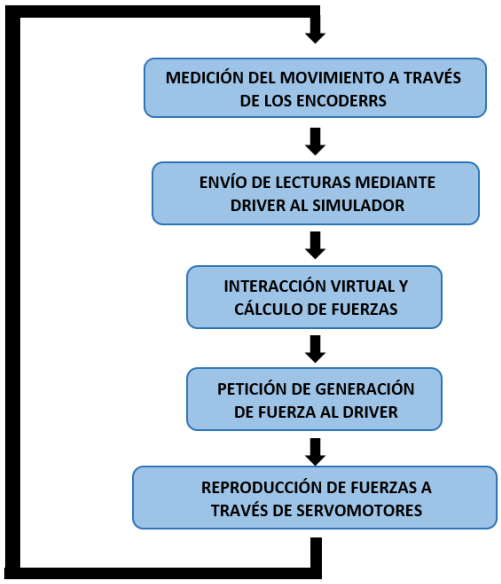


Figura 11. Diagrama del flujo lógico de un dispositivo háptico

Un parámetro importante es el espacio de trabajo nominal, que es el volumen en el cual el fabricante garantiza la precisión de fuerza especificada.

El espacio de trabajo real, sin embrago, se define como el volumen que es posible recorrer con el dispositivo incluyendo zonas de mínima precisión.

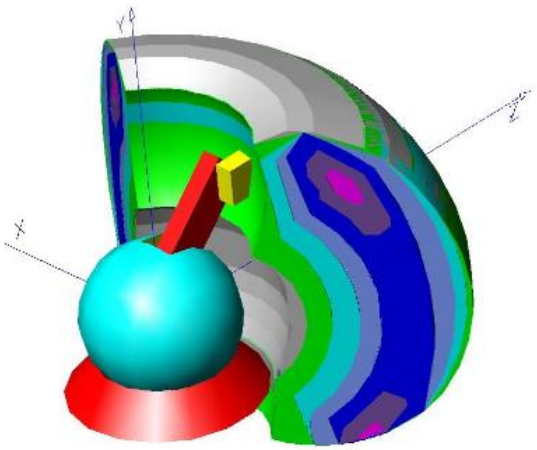


Figura 12. Volumen de trabajo de los dispositivos hápticos

Claro está que ante estos dos espacios se nos plantea un tercero que es el espacio de trabajo de la aplicación, siendo el volumen que se está simulando. Es el único de los tres que es diferente para cada simulación y se ha de tener en cuenta a la hora de implementar el código que haga que los tres coincidan de la mejor forma posible.

La parte clave para una simulación de un programa con dispositivos hápticos es, sin duda una correcta percepción de las fuerzas que se debe ejercer para que el usuario sienta fluidez y realismo durante los movimientos. De seguido, se tratarán esos puntos imprescindibles que se deben cumplir en el diseño de un para la correcta percepción y representación de dichas fuerzas.

1.3.1 PERCEPCIÓN Y REPRESENTACIÓN DE FUERZAS CON DISPOSITIVOS HÁPTICOS

La representación de fuerzas mediante dispositivos hápticos es en cierto modo análoga a la representación gráfica de animaciones. Se trata de hacer percibir una sensación externa a través de la generación de estímulos sucesivos. Los dispositivos hápticos generan fuerzas miles de veces por segundo, de tal manera que un usuario pueda notar la sensación de tocar a un modelo virtual. Aunque en la realidad no se trate de una acción constante, si no que se realiza a una frecuencia tan alta que el cerebro se ve engañado y la percibe como si lo fuera.

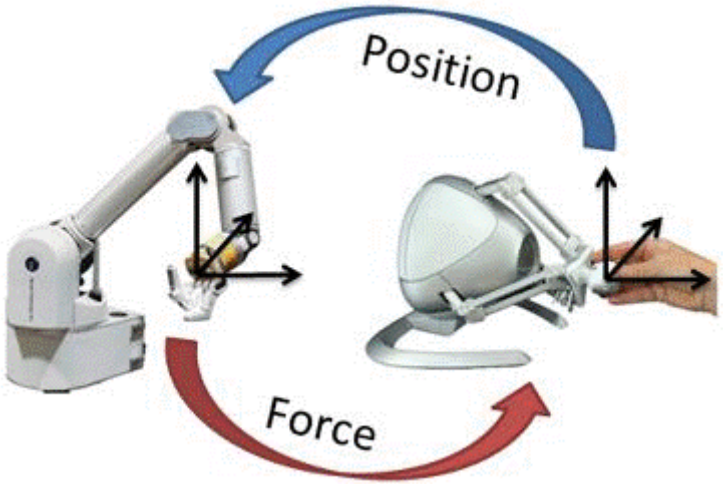


Figura 13. Feedback fuerza posición

Las principales claves para lograr que esta ilusión sea percibida como real, y así obtener una buena interfaz háptica, son:

- Rango de representación de fuerzas ideal: nulo cuando el movimiento es libre, pero adecuado (en dirección y magnitud) cuando se toca o agarra un objeto en el entorno virtual.

- La interfaz debe operar dentro de las habilidades y limitaciones de una persona.

- Sensación de continuidad: continuando con el símil de la generación de animaciones mediante la exposición a imágenes rápidamente cambiantes, el sistema háptico tiene que ser capaz de representar cambios en las fuerzas representadas con la suficiente continuidad como para ser percibidos como continuos. A diferencia de las animaciones, donde una frecuencia de refresco a partir de 24Hz es considerada continua, se requiere una tasa de actualización de al menos 1000Hz para que no se perciban golpes o discontinuidades.

- Intervalo, resolución y ancho de banda adecuados. El usuario no debe ser capaz de poder atravesar objetos rígidos por exceso de rango de fuerza (Incluye parada mecánica). No puede haber vibraciones involuntarias. Los objetos sólidos deben sentirse.

- La magnitud y la tasa de cambios no son especialmente relevantes, dado que es difícil percibir pequeños incrementos continuos en los valores de las fuerzas. El hecho de que el dispositivo tenga una limitación en la magnitud de representación de la fuerza no impide que éstas se puedan escalar y conseguir los efectos deseados.

En el caso, por ejemplo, de un ensamblaje interactivo como el que se realiza en el montaje a partir de unas piezas diseñadas en SolidWorks; sería muy interesante la aplicación del dispositivo háptico que complemente con un sentido a mayores la realidad virtual.

2. DISEÑO DEL SIMULADOR

Los objetivos que se esperan alcanzar con este estudio son:

- El diseño de un simulador capaz de manipular objetos diseñados en un software de CAD con respuesta háptica.
- El empleo de varios dispositivos hápticos en la misma simulación de forma cooperativa.
- Poder realizar un ensamblaje con la ayuda únicamente de la respuesta háptica de los dispositivos y la visualización del montaje.
- Demostrar la utilidad de este simulador como complemento de programas de CAD actuales.

Para el diseño del simulador será necesario aclarar los conceptos que se detallan a continuación.

2.1 COMPUTACIÓN GRÁFICA 3D

La computación gráfica o gráficos por ordenador es el campo de la informática visual, donde se utilizan computadoras tanto para generar imágenes visuales sintéticamente como integrar o cambiar la información visual y espacial probada del mundo real.

Los polígonos tridimensionales son la base de prácticamente todos los gráficos 3d realizados en computadora. Como consiguiente, la mayoría de los motores de gráficos de 3D están basados en el almacenaje de puntos (por medio de 3 simples coordenadas Dimensionales X,Y,Z), líneas que conectan aquellos grupos de puntos, las caras son definidas por las líneas, y luego una secuencia de caras crean los polígonos tridimensionales.

Una malla de polígono es una colección de vértices, bordes y caras que define la forma de un objeto poliédrico en gráficos de computadora 3D y modelado sólido. Las caras suelen consistir en triángulos (malla triangular), cuadriláteros u otros polígonos convexos simples, ya que esto simplifica la representación, pero también puede estar compuesto de polígonos cóncavos más generales o polígonos con agujeros.

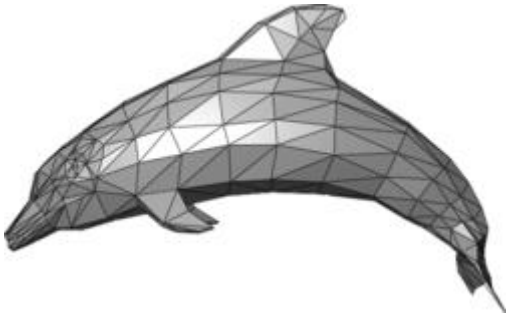


Figura 14. Mallado por triángulos de un objeto

La variedad de operaciones realizadas en las mallas puede incluir lógica booleana, suavizado, simplificación y muchas otras.

Las mallas volumétricas son distintas de las mallas poligonales, ya que representan explícitamente tanto la superficie como el volumen de una estructura, mientras que las mallas poligonales representan únicamente explícitamente la superficie (el volumen está implícito).

Como las mallas poligonales se utilizan ampliamente en gráficos por ordenador, también existen algoritmos para el trazado de rayos, la detección de colisiones y la dinámica de cuerpos rígidos de mallas poligonales.

Una malla de triángulo es un tipo de malla de comprendida por un conjunto de triángulos (típicamente en tres dimensiones) que están conectados por sus bordes o esquinas comunes.

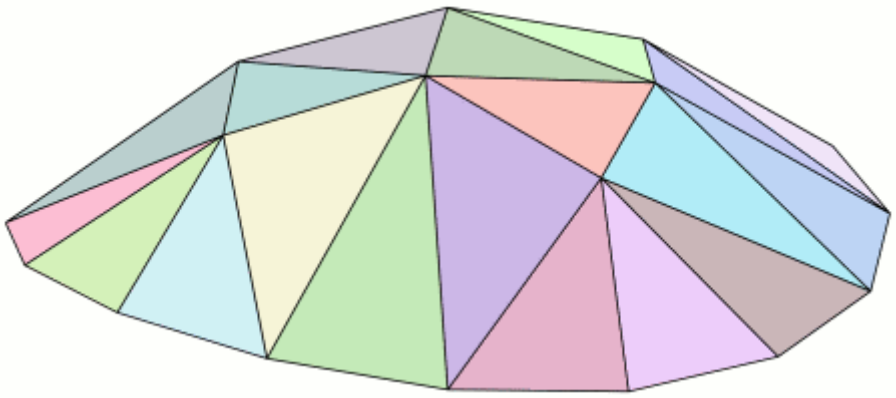


Figura 15. Malla de triángulo

Muchos paquetes de software de gráficos y dispositivos de hardware pueden operar más eficientemente en triángulos que se agrupan en mallas que en un número similar de triángulos que se presentan individualmente. Esto es típicamente porque los gráficos de computadora hacen operaciones en los vértices en las esquinas de triángulos. Con triángulos individuales, el sistema tiene que operar en tres vértices para cada triángulo. En una malla grande, podría haber ocho o más triángulos reunidos en un solo vértice procesando esos vértices sólo una vez, es posible hacer una fracción de la tarea y lograr un efecto idéntico.

En muchas aplicaciones de gráficos por ordenador es necesario administrar una malla de triángulos. Los componentes de malla son vértices, bordes y triángulos.

Una aplicación puede requerir el conocimiento de las diversas conexiones entre los componentes de malla. Estas conexiones se pueden gestionar independientemente de las posiciones reales de los vértices.

2.2 RETROALIMENTACIÓN DE FUERZAS (OPEN HAPTICS)

En primer lugar, es necesario aclarar que, para la creación de un programa es necesario el cumplimiento de una serie de tareas:

- La creación de una ventana (que es el área visual, normalmente de forma rectangular, que contendrá algún tipo de interfaz de usuario, mostrando la salida y permitiendo la entrada de datos para uno o varios procesos que se ejecutan simultáneamente).
- La gestión del teclado/ratón (que son los periféricos que permitirán la entrada de información por parte del usuario hacia el programa), aunque el principal dispositivo que introducirá la información es el dispositivo háptico
- Y para este caso el control sobre los dispositivos hápticos.

Para el control de los dispositivos hápticos a través del PC, es necesario establecer una comunicación continua entre ambos sistemas. Existen librerías para esta tarea, una de las cuales es OpenHaptics, que es la proporcionada por el fabricante del dispositivo.

OpenHaptics es fruto de la colaboración de varios fabricantes de dispositivos hápticos con la esperanza de tener una interfaz común para su programación independientemente del modelo de dispositivo del que se trate.

Con OpenHaptics se puede controlar y programar cualquiera de las características de un dispositivo concreto como su configuración de posición o bien la magnitud y el sentido de las fuerzas que se desean generar.

Mencionar que proporcionan una serie de ejemplos en su manual como: carga de objetos y eventos callback (picked apples), distintas vitas de ventanas (multiplewindow), propiedades de cuerpos deformables (spongycow), implementación de fuerzas (skullcoulombforce), renderizado de modelos hápticos (shapedepthfeedback).

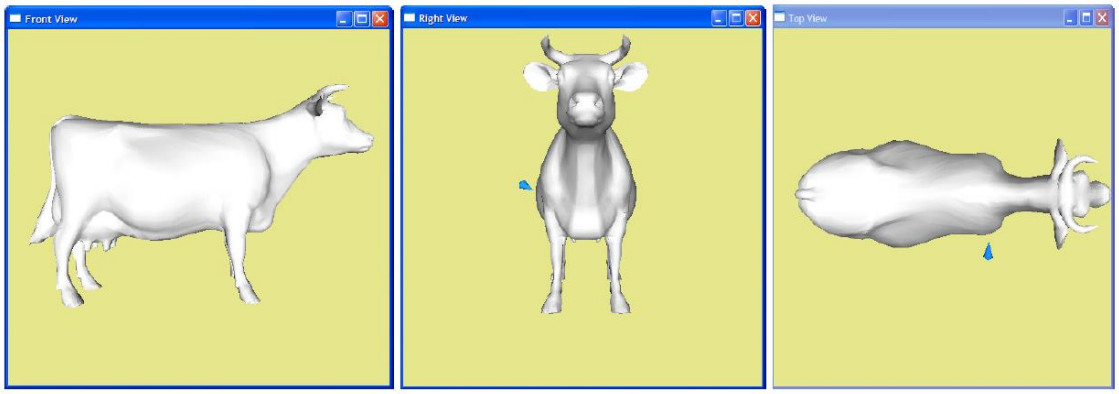


Figura 16. Ejemplo multiplewindow de Open Haptics

OpenHaptics son una serie de librerías ordenadas a modo de lenguajes de dsitinto nivel de programación: QuickHaptichs, HL y HD.

Son APIs que permiten programar los dispositivos hápticos. A continuación, se describirán ordenadas de más alto nivel a más bajo. Obvia decir que, en cualquier momento se puede emplear una interfaz de más bajo nivel a la que se estaba usando.

QuickHaptics proporciona librerías para la creación de múltiples ventanas, carga de ficheros de malla y primitivas de colisión para desarrollo rápido.

HL crea dos hilos (threads) de ejecución, el servo-loop (1000Hz) y el de los callbacks de colisión (100Hz), (un callback es una función "A" que se usa como argumento de otra función "B". Cuando se llama a "B", ésta ejecuta "A". Para conseguirlo, usualmente lo que se pasa a "B" es el puntero a "A"). HL define las mallas de manera muy similar a OpenGL (todo este tema será tratado posteriormente), de tal manera que se pueda compartir la implementación. Tiene 3 modos, entre ellos un modo «proxy» con el que se puede simular un objeto virtual. Se pueden emplear «materiales».

HD es la API de más bajo nivel en la que se programan directamente las fuerzas que se quieren representar.

- Callback asíncrono: función que se ejecuta en cada paso (tick) del planificador (scheduler) a la frecuencia que oscila entre 500 y 2000Hz.

- Callback síncrono: función que se ejecuta una vez, y por la que el hilo principal del programa espera. Se emplea para consultar datos instantáneos sin que haya conflictos entre ambos hilos.

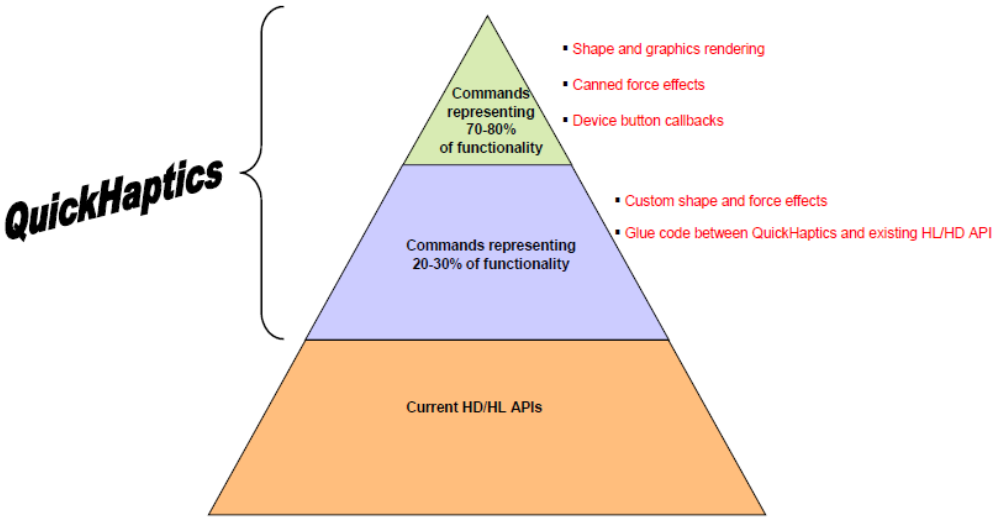


Figura 17. Pirámide de niveles de las librerías OpenHapics

Para un programa de estas características y distintas frecuencias de testeo para tareas que se deben correr simultáneamente, se recurre al uso de hilos o threads.

2.2.1 Programación multihilo (multithread)

En sistemas operativos, un hilo de ejecución, hebra o subproceso es una secuencia de tareas encadenadas muy pequeña que puede ser ejecutada por un sistema operativo.

Los distintos hilos de ejecución comparten una serie de recursos tales como el espacio de memoria, los archivos abiertos, la situación de autenticación, etc. Esta técnica permite simplificar el diseño de una aplicación que debe llevar a cabo distintas funciones simultáneamente, aunque requieran diferentes tasas de actualización (por ejemplo, el bucle del háptico a 1000Hz y el del programa a ~60Hz).

Un hilo es simplemente una tarea que puede ser ejecutada al mismo tiempo que otra tarea.

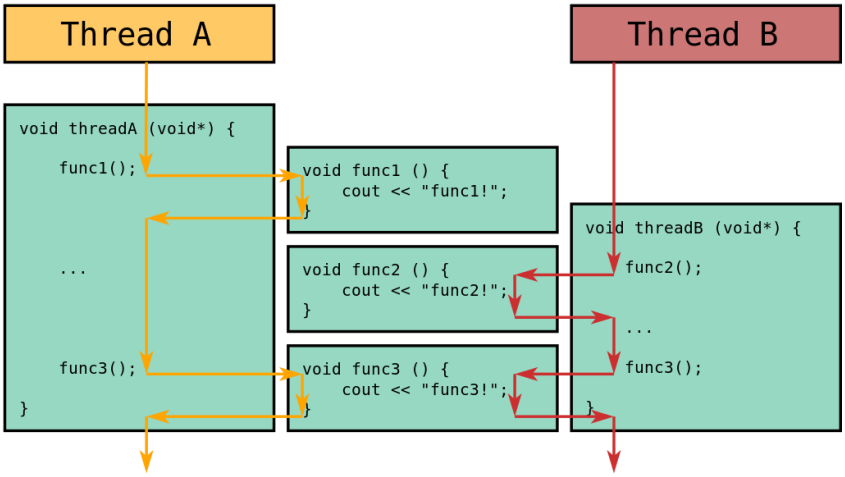


Figura 18. Ejemplo multihilo

Los hilos de ejecución que comparten los mismos recursos, sumados a estos recursos, son en conjunto conocidos como un proceso. El hecho de que los hilos de ejecución de un mismo proceso compartan los recursos hace que cualquiera de estos hilos pueda modificar estos recursos.

Cuando un hilo modifica un dato en la memoria, los otros hilos acceden a ese dato modificado inmediatamente. Debido a este motivo, es muy importante no modificar variables desde distintos hilos, ya que los valores de estas podrían ser inconsistentes, por ejemplo, en el caso de estar escribiendo una matriz en un hilo y leyendo en otro, las componentes de la matriz que se leyeran no serían las que se pretenden.

El proceso sigue en ejecución mientras al menos uno de sus hilos de ejecución siga activo. Cuando el proceso finaliza, todos sus hilos de ejecución también lo han hecho. Asimismo, en el momento en el que todos los hilos de ejecución finalizan, el proceso deja de existir y se liberan todos los recursos.

OH ya de por sí segmenta las tareas de representación gráfica en un número de hilos que dependerá de la capa de abstracción que se emplee (QH, HL, HD).

Dado que la renderización háptica requiere actualizaciones más frecuentes que las aplicaciones gráficas típicas, el motor de renderización crea, además del hilo principal de la aplicación, dos hilos adicionales que utiliza para el renderizado háptico: el *servoloop* o ´´hilo servo`` y el hilo de colisión.

El hilo principal de la aplicación en un programa se denomina client thread o ´´hilo cliente``. El hilo cliente es el hilo en el que se crea el contexto de renderización y en el que las funciones son llamadas por los programas cliente. El código se escribirá para ejecutarse en su subproceso de cliente sin necesidad de conocer sobre que hilo se trata, aunque hay casos en los que sea necesario la ejecución en uno de estos subprocesos.

El hilo de alta prioridad (servoloop) tiene un rango de frecuencias entre 500Hz y 2000Hz, este maneja la comunicación directa con el dispositivo háptico. Lee la posición y orientación del dispositivo y actualiza la fuerza enviada al dispositivo a una velocidad alta (usualmente 1000 hz). Este hilo se ejecuta en una prioridad elevada con el fin de mantener una representación háptica estable. Este hilo solo es visible en el caso del HDAPI, HLAPI y QH oculta el servo del usuario (con la excepción de efectos de fuerza personalizados).

Cabe mencionar que el comportamiento del servoloop es automático con algunas librerías y en otros casos puede ser definido por el usuario.

El hilo de cálculo de efectos de fuerza en función de detección de colisiones (100Hz) es responsable de determinar qué primitivas geométricas están en contacto con el proxy. Funciona a una velocidad de 100 hz que es más lento que el hilo del servo, pero más rápido que el hilo del cliente.

El hilo de colisión encuentra cuál de las formas especificadas en el hilo del cliente están en contacto con el proxy y genera aproximaciones locales simples de esas formas. Estas aproximaciones locales son enviadas al hilo servo que los utiliza para actualizar la fuerza al dispositivo háptico.

El uso de funciones locales sencillas permite que el hilo servo mantenga una alta tasa de actualización incluso si el número de primitivas geométricas proporcionadas es alta. Cabe mencionar que en nuestro estudio no tuvo aplicación este hilo debido a que para el estudio de colisiones se empleó Bullet. A pesar de que el sistema de cáculo dinámico empleado, ya incluía esta funcionalidad, y se beneficiaba de una mayor integración. Sin embargo, no estaba preparado para colisiones de múltiples objetos con múltiples punteros hápticos. Por este motivo se decidió cambiar a otro.

2.2.1.1 Bucle de representación de fuerzas (ServoLoop)

El *ServoLoop* se refiere al bucle de control usado para calcular fuerzas para enviar al dispositivo háptico. Con el fin de renderizar una respuesta estable, este bucle se debe ejecutar a una tasa de 1khz consistente o mejor. Con el fin de mantener una tasa de actualización tan alta, el bucle de servo es generalmente ejecutado en un hilo separado, de alta prioridad.

A continuación, se muestra el diagrama de flujo empleado en un programa de Open Haptics y en él se puede ubicar este servoloop. que sirve como canal de información entre HDAPI y el dispositivo conectada.

Es necesario distinguir entre el servoloop y el graphicsloop o ´´bucle de representación gráfica``. Este a diferencia del servoloop se actualiza a una frecuencia menor, ya que no es necesario un testeo tan rápido para la representación visual.

Esto es debido a que el sentido del tacto es más difícil de engañar que el de la vista. Y, por tanto, necesita una velocidad mayor de transmisión de datos.

El bucle de hilo principal o servoloop se ejecuta en paralelo con otros hilos. Para que el programa sepa a qué hilo prevalezca sobre otro y evitar conflictos entre ellos, es necesario definir en que consiste la prioridad de un hilo.

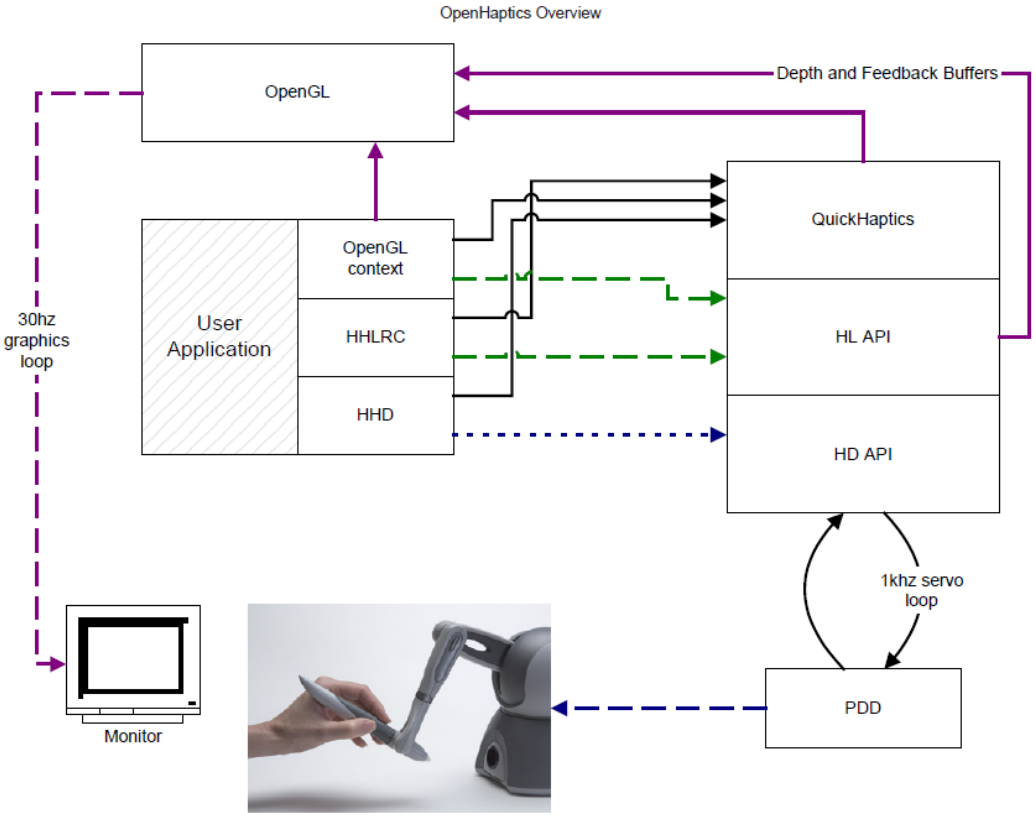


Figura 19. Diagrama funcionamiento OH con servoloop

La prioridad de un hilo indica la preferencia que tiene un hilo sobre otro. Se utiliza para decidir cuándo pasar a ejecutar otro hilo (cambio de contexto).

- Cuando un hilo cede el control (por abandono explícito o por bloqueo), se ejecuta a continuación el que tenga mayor prioridad.
- Un hilo puede ser desalojado por otro con prioridad más alta, tan pronto como este desee hacerlo.

Pero en la práctica la cantidad de CPU que recibe cada hilo depende además de otros factores como la forma en que el sistema operativo implementa la multitarea.

El *scheduler* o ´´programador`` gestiona un hilo de alta prioridad para enviar fuerzas y recuperar información del estado del dispositivo. Normalmente, las actualizaciones de la fuerza necesitan ser enviadas a una frecuencia de 1000 Hz para crear una respuesta de fuerza convincente y estable. El scheduler permite que la aplicación se comunique de forma efectiva con el subproceso del bucle de una manera segura, y agregue operaciones. Las diferentes tareas que puede gestionar son:

- Tareas asíncronas: típicamente se ejecutan una vez por ciclo y de manera continua, independientemente del resto del programa.

- Tareas síncronas: se ejecutan sólo una vez, y el servoloop espera a que hayan finalizado. Se emplean principalmente para transferir información entre el ServoLoop y el resto del programa. Porque en ese momento varios hilos no pueden estar modificando los mismos datos (posición, fuerza). Es decir, como las variables de estado y datos se comparten puede haber problema de inconsistencias en su lectura o escritura cuando varios hilos tratan de acceder a un mismo dato simultáneamente. Estas tareas permiten el acceso a esas distintas frecuencias que tendrán los hilos.

La sincronización de estado se vuelve importante cuando se administra una interfaz de usuario que incluya tanto gráficos hápticos como gráficos, porque hay dos bucles de renderización que necesitan acceso a la misma información. Esto suele implicar la realización de copias seguras de los datos cada subproceso como una instantánea de estado.

La utilización de la exclusión mutua como una técnica de sincronización, puede introducir errores fácilmente. Además, el bucle de renderizado debe mantener una frecuencia de 1000Hz, y por lo tanto no debería espera nunca por otro de prioridad inferior para liberar un bloqueo, especialmente si está realizando otras llamadas al sistema que lo bloqueen por un tiempo indefinido.

También hay que tener en cuenta la disparidad de actualización entre el bucle gráfico y el bucle háptico, que haría muy fácil mostrar un estado inconsistente si varios objetos se movieran en la pantalla al mismo tiempo.

Por todo esto es recomendable tratar la sincronización de estado entre los subprocesos como una operación de copia instantánea de acceso a datos de forma discontinua, para ello el empleo de la función Mutex.

2.2.2 Programación orientada a eventos con OpenHaptics

En ciertas ocasiones es posible no manejar directamente el ServoLoop, y programar únicamente funciones que definen el comportamiento que debe presentar el dispositivo ante situaciones previstas (son los llamados *callbacks* que se abordarán más adelante): pulsación de botones, detección de contacto contra otros objetos, etc.

2.2.2.1 Eventos de QuickHaptics

QuickHaptics es una micro API facilita el escribir nuevas aplicaciones o añadir hápticos a las aplicaciones existentes. Los analizadores de geometría y los parámetros por defecto inteligentes hacen posible configurar escenas hápticas / gráficas sin necesidad de llegar a usar un lenguaje excesivamente bajo de programación.

2.2.2.2 Eventos de HL

Tradicionalmente se representan objetos gráficos en pantalla mediante la definición de las superficies formadas a partir de mallas de triángulos. Dado que el problema de detección de contactos emplea la misma información, la librería HL permite aplicar modelos de fuerzas de contacto directamente a partir de las mismas definiciones.

2.2.2.3 Callbacks

Definimos callbacks como funciones que se llaman como resultado de un evento, son funciones de argumentos ya establecidos para modificar el algoritmo de un código existente, en este caso la librería.

Esto permite desarrollar capas de abstracción de código genérico a bajo nivel que pueden ser llamadas desde una subrutina (o función) definida en una capa de mayor nivel. Usualmente, el código de alto nivel inicia con la llamada de alguna función, definida a bajo nivel, pasando a esta un puntero, o un puntero inteligente (conocido como handle), de alguna función.

Mientras la función de bajo nivel se ejecuta, esta puede ejecutar la función pasada como puntero para realizar alguna tarea. En otro escenario, las funciones de bajo nivel registran las funciones pasadas como un handle y luego pueden ser usadas de modo asincrónico.

Un callback puede ser usada como una aproximación simple al polimorfismo y a la programación genérica, donde el comportamiento de una función puede ser dinámicamente determinado por el paso punteros a funciones o handles a funciones de bajo nivel que, aunque realicen tareas diferentes los argumentos sean compatibles entre sí. Esta es una técnica de mucha importancia por lo que se la llama código reutilizable.

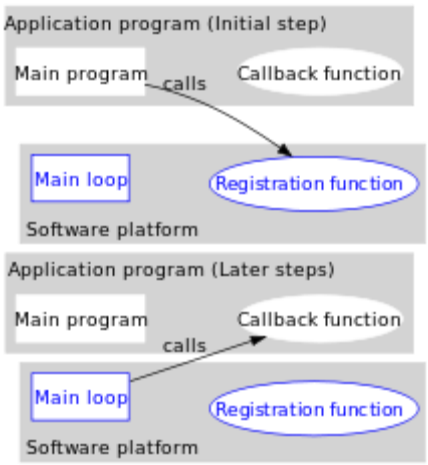


Figura 20. Esquema del funcionamiento de un callback

Se puede considerar como ejemplo el problema de realizar varias operaciones arbitrarias en una lista. Una opción puede ser iterar sobre la lista, o también realizar alguna operación sobre cada uno de los elementos de la lista. En la práctica, la solución más común, pero no ideal, es utilizar iteradores como un bucle for, que deberá duplicarse en cada lugar del código donde sea necesario. Más aún, si la lista es actualizada por un proceso asíncrono (por ejemplo, si un elemento es añadido o eliminado), el iterador podría corromperse durante el paso a través de la lista.

Una alternativa podría ser crear una nueva biblioteca de funciones que ejecute la tarea deseada con la sincronización apropiada en cada caso. Esta propuesta aún requiere que cada nueva función de la biblioteca contenga el código para ir a través de la lista. Esta solución no es aceptable para bibliotecas genéricas que tengan como objetivo varias aplicaciones; el desarrollador de la biblioteca no puede anticiparse a las necesidades de cada aplicación, y el desarrollador de las aplicaciones no debería necesitar conocer los detalles de la implementación de la biblioteca.

En este caso los callbacks resuelven estos problemas. Un procedimiento es escribir el paso a través de una lista que provee a la aplicación del código para ir a través de la lista y operando sobre cada elemento. Existe una clara distinción entre la biblioteca y la aplicación sin sacrificar la flexibilidad. Un callback también considerarse un tipo de rutina enlazada por referencia.

El puntero, un contacto entre objetos o una pulsación de botón son ejemplos de eventos que pueden activar un callback, debido a que en la librería no puede venir establecido lo que el usuario desea que ocurra en esos casos.

Existen tres modos de generar un callback usando el HLAPI:

1. *Shape rendering* o renderizado de la forma: permite especificar la geometría de primitivas, que será empleada por el motor de renderizado para calcular las reacciones y fuerzas necesarias de simulación del contacto con la superficie.

2. *Effect rendering* o renderizado de efectos: permite definir funciones globales de fuerza, que no son fácilmente determinadas por una geometría o que no tendrían sentido (una mera vibración no es comparable con un contacto) y es posible generarlas en cualquier posición del dispositivo háptico. HLAPI incluye algunos efectos de fuerzas estándar como resistencia al movimiento de tipo viscoso y resortes en funciones programables directamente.

3. *Direct proxy rendering* o representación de proxy directo: permite establecer una posición y orientación deseada para el dispositivo háptico, generando automáticamente las fuerzas apropiadas para llevar al dispositivo a dicha posición. Debido a que, en este estudio, la detección de colisiones y contacto entre cuerpos la lleva a cabo Bullet, es necesario emplear este tipo de callback que permite tratar la posición y fuerza necesaria en cada momento de forma explícita.

2.2.3 Implementación de fuerzas deseadas

La función startServoLoopCallback define los elementos necesarios para implementar un efecto de fuerza deseada en HL. Toma como parámetros tres punteros de funciones.

- startEffectCB
- computeForceCB
- stopEffectCB

2.2.3.1 StartEffectCB y stopEffectCB

startEffectCB es invocada una vez iniciado el servoloop y stopEffectCB es invocada una vez se haya salido del bucle. Entre estes dos callbacks, computeForceCB es llamada repetidamente a una frecuencia de 1kHz, es decir, a la misma que el servoloop.

Debido a lecturas erróneas, que el dispositivo detecta automáticamente, este responde reiniciando los valores internos del dispositivo y ofrece al usuario la posibilidad de reiniciar el estado de sus modelos de fuerzas mediante el callback startEffectCB. Esto es especialmente relevante cuando los valores de las fuerzas se computen en función de valores previos en el tiempo.

2.2.3.2 ComputeForceCB

Esta función recibe tres parámetros de OpenHaptics:

1. HDdouble force[3]
2. HLcache *cache
3. Void *userdata

La representación de fuerzas del punto de contacto del dispositivo activo la lleva a cabo force.

*cache es un puntero a las fuerzas almacenadas anteriores y otros valores de propiedad justo antes del marco actual del servo loop.

Void *userdata es un puntero void que apunta a cualquier nuevo dato que el usuario desee pasar a OpenHaptics a través de esta devolución de llamada. En esencia, este callback lee los valores de fuerza actuales, los modifica usando los valores señalados por void * userdata y envía las nuevas fuerzas al dispositivo háptico.

2.3 CONEXIÓN DE MÚLTIPLES DISPOSITIVOS HÁPTICOS CON OPEN HAPTICS

El simulador puede soportar el uso compartido de múltiples dispositivos hápticos. Para manifestar la presencia de un dispositivo y activarlo es necesario comenzar explicando lo que es un contexto.

Un contexto refleja el estado interno de un dispositivo desde que se inicializa, todas las comunicaciones desde y hacia el dispositivo se realizan en Open Haptics a través de esos contextos a modo de identificador.

Cada contexto de representación háptica generará fuerzas para cada dispositivo, por lo que, para utilizar varios dispositivos, se utilizará un contexto de representación independiente para cada dispositivo.

Es posible obtener de cada dispositivo una serie de parámetros mientras este esté activo. En la siguiente figura se muestran algunos parámetros que son posibles obtener mediante una llamada a función de HL:

Nombre de parámetro	Descripción	Número de argumentos	Tipos permitidos
HD_CURRENT_BUTTONS	Obtener el estado del botón. Los valores de botones individuales se pueden recuperar haciendo bit a bit y con HD_DEVICE_BUTTON_N (N=1,2,3, o 4).	1	HDint
HD_CURRENT_ENCODER_VALUES	Obtener los valores del encoder sin procesar.	6	HDlong
HD_CURRENT_POSITION	Obtener la posición actual del dispositivo frente a la base del dispositivo. Derecha es + x, arriba es +y, hacia el usuario es + z.	3	HDdouble, HDfloat
HD_CURRENT_VELOCITY	Obtener la velocidad actual del dispositivo. Nota: Este valor se suaviza para reducir la fluctuación de alta frecuencia.	3	HDdouble, HDfloat
HD_CURRENT_TRANSFORM	Obtener la transformada de la posición del dispositivo.	16	HDdouble, HDfloat
HD_CURRENT_ANGULAR_VELOCITY	Obtener la velocidad angular del dispositivo.	3	HDdouble, HDfloat
HD_CURRENT_GIMBAL_ANGLES	Obtener los ángulos del dispositivo. Para dispositivos táctiles: Desde la posición Neutral La derecha es +, arriba es -, CW es +	3	HDdouble, HDfloat
HD_USER_STATUS LIGHT	Obtenga la configuración de usuario de la luz LED de estado. Vea hdDefine.h para los ajustes constantes.	1	HDint

Figura 21. Tabla de parámetros que son posibles obtener en un callback de HL

Durante el inicio del programa, primero se inicializan todos los dispositivos hápticos y se obtienen referencias de cada uno de ellos.

A continuación, se crea un contexto de renderizado háptico para cada dispositivo. Es importante tener en cuenta que el planificador se inicia automáticamente cuando se crea un contexto de renderizado háptico con un dispositivo válido, por lo que son inicializados después iniciar el planificador, y no participarán en el planificador a menos que el planificador sea apagado y luego encendido. Por lo tanto, el orden es inicializar todos los dispositivos antes de crear su respectivo contexto háptico de representación.

El renderizado háptico para múltiples dispositivos y contextos implica ejecutar un pase de representación por separado para cada contexto. El uso de la función hlMakeCurrent () se usa para orientar las operaciones de renderizado en un contexto y sólo sería posible la comunicación con un dispositivo en particular, el que esté activo en ese momento.

Esto es lo conveniente, debido a que cada geometría requerida para la ubicación es particular para cada dispositivo. A partir de esto, se hace necesario en el callback conocer el dispositivo que se está tratando en cada llamada, es posible preguntando que dispositivo está activo con la función hdgetCurrentDevice().

También se comienza un marco háptico y se comprueban los eventos de todos los contextos para que la entrada y los eventos puedan ser procesados para todos los dispositivos antes de representar nuevas formas y efectos para cada uno de ellos.

2.4 CALIBRACIÓN

Los dispositivos se calibran después de haber sido inicializados. Algunos dispositivos requieren calibración manual ya que necesitan un reinicio hardware de los encoders.

La calibración permite al dispositivo conocer su posición física de forma precisa. Por ejemplo, antes de llamar a la calibración, un dispositivo podría pensar que su brazo está en el centro del espacio de trabajo, mientras que el brazo está en realidad a un lado. Hay algunos métodos diferentes para realizar la calibración, dependiendo de su tipo de calibración:

a) Reajuste hardware de los encoders: En esta forma de calibración, el usuario coloca manualmente la unidad en una posición de reposo y llama a la rutina de calibración. Para muchos dispositivos, la posición de reposo es tal que todos los ángulos del brazo son ortogonales. Normalmente, esto sólo se debe realizar una vez cuando se conecta la unidad y la calibración persistirá hasta que se desenchufe la unidad o se realice otra restauración del hardware.

b) Calibración del tintero: En esta forma de calibración, el usuario coloca el lápiz en el tintero, que limita tanto su posición como su orientación, y llama a la rutina de calibración. Normalmente, el planificador debe estar ejecutándose para que esta forma de calibración tenga éxito. Esta calibración persiste entre distintas sesiones.

c) Calibración automática: En la calibración automática, el dispositivo utiliza internamente mecanismos para actualizar su calibración a medida que se

mueve la unidad. Este estilo también soporta calibración parcial en la que se obtiene información sobre uno o más ejes de tal manera que la calibración se puede realizar a lo largo de esos ejes. Esta calibración persiste de una sesión a otra.

Dado que cada tipo de dispositivo tiene una forma diferente de calibración, el tipo de calibración soportado se puede consultar a través de la función HD_CALIBRATION_STYLE, que devuelve el tipo de calibración soportada por el dispositivo. Para el reinicio del encoder y la calibración del tintero, se debe solicitar al usuario que coloque la unidad en la posición que corresponda y hdUpdateCalibration () se debe llamar una vez. Para calibración automática, la calibración se debe verificar periódicamente con HdCheckCalibration () y actualizar mediante hdUpdateCalibration (). Por ejemplo, el tipo de calibración necesaria para los dispositivos empleados en este estudio es la de calibración en el tintero.

¿Cuándo calibrar?

Dado que la calibración puede hacer que la posición del dispositivo salte, la calibración debe realizarse normalmente con un cierto control de la fuerza, o desactivación de fuerzas. De lo contrario, por ejemplo, la calibración podría hacer que el dispositivo saltara en una posición donde una gran fuerza sería devuelta en respuesta (como el interior de un objeto).

Es conveniente mencionar que Open Haptics proporciona unas rutinas para el control de seguridad, como, por ejemplo, sobrecargas, sobrevelocidades, rampas de fuerza y control de la temperatura.

Algunos de estos mecanismos de seguridad están controlados por el hardware del dispositivo y no pueden ser sobrescritos.

2.5 MODELOS HÁPTICOS DE FUERZAS DE CONTACTO

Evaluar la fuerza necesaria para representar un contacto en un dispositivo háptico, no es un problema de solución trivial.

La primera solución que cabe pensar ante esta situación es crear un objeto virtual que sea movido por el háptico directamente, sin embargo, la simulación llevaría a entradas inconsistentes para la resolución de la simulación en cuanto el puntero del háptico llegase a posiciones inalcanzables en la simulación

Por ejemplo, el caso de la violación de la condición de distancia constante entre puntos de un sólido rígido, como cuando un objeto intenta penetrar a otro.

Para obtener unas fuerzas que se parezcan a las que sucederían si tocásemos los objetos de forma real: habrá que distinguir entre el HIP (haptic interface point) representación del dispositivo en el mundo virtual y el PROXY que es la representación del dispositivo virtual, el cual coincidirá con el HIP cuando no exista colisión.

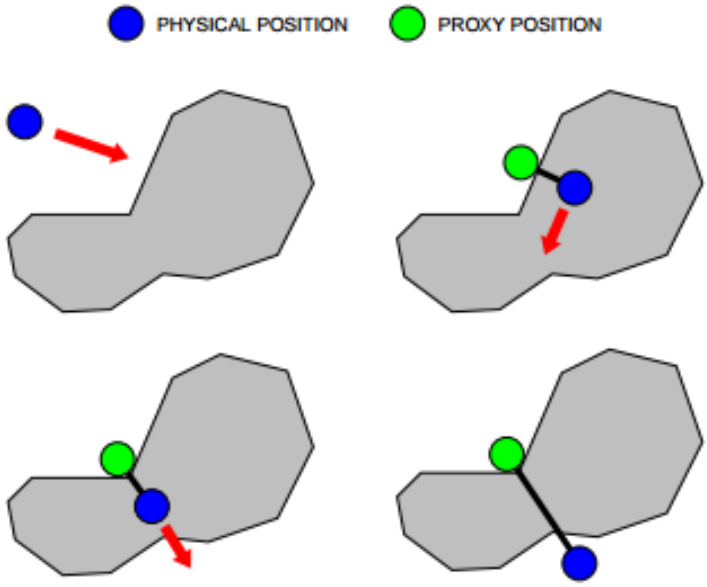


Figura 22. Representación PROXY e HIP

Usaremos esta distancia entre ellos para evaluar la fuerza a aplicar de reacción en el dispositivo. Una primera aproximación podría ser a partir de la ley de Hooke (usando un modelo lineal).

Donde el valor máximo de la K viene dado por el límite en magnitud de representación de fuerza, por eso hay que escalar las magnitudes al rango factible. Está sensación de cambios bruscos en la fuerza podrían mejorarse con un factor amortiguador que dependa de la velocidad, aunque en este estudio no ha sido necesario.

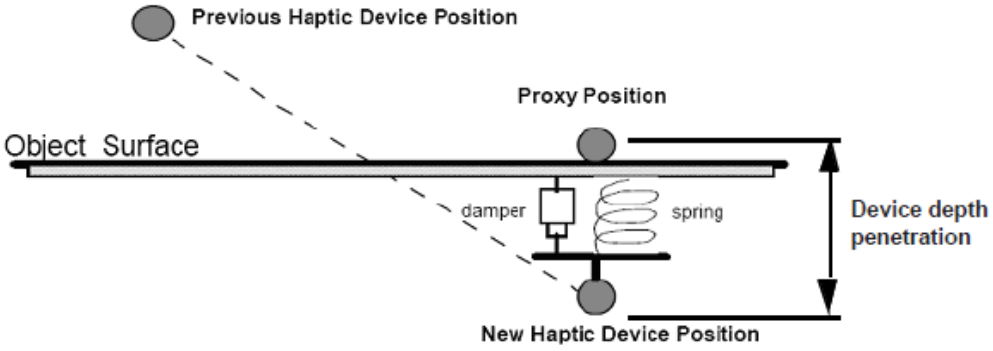


Figura 23. Situación del puntero durante un contacto

Durante el estudio inicialmente se producían inestabilidades causadas por brusquedad en las fuerzas. Dichas brusquedades se producen cuando entre un instante y el siguiente hay una gran distancia entre las posiciones del puntero. Se ha optado por una solución que consiste en limitar a un valor máximo la fuerza calculada a través de la

distancia entre el puntero del dispositivo y el puntero virtual de la simulación, de tal manera que ambas converjan de un modo más suave.

2.5.1.1 Cálculo del PROXY

Existen varios métodos para el cálculo de distancia entre el HIP y el PROXY de dos objetos en contacto. Uno de ellos consiste en un proceso iterativo que averigüe el punto más cercano sobre la superficie del objeto. Para ello se divide el objeto mediante un diagrama de voronoi.

Este diagrama es una de las estructuras fundamentales en la geometría computacional, esta almacena toda la información referente a la proximidad entre puntos.

La idea del diagrama de Voronoi se basa fundamentalmente en la proximidad. Se supone dado un conjunto finito de puntos en el plano P = {p1,...,pn} (con n mayor o igual que dos) y a cada pj se le asocia aquellos puntos del plano que están más cerca o igual suya que de cualquier otro de los pi con i distinto de j. Todo punto del plano queda así asociado a algún pi, formándose conjuntos que recubren a éste. Existirán puntos que disten lo mismo de dos elementos de P y que formarán la frontera de cada región. Los conjuntos resultantes forman una teselación del plano, en el sentido de que son exhaustivos (todo punto del plano pertenece a alguno de ellos) y mutuamente excluyentes salvo en su frontera. Se le llama a esta teselación: Diagrama de Voronoi plano (denotado Vor(P)). A cada una de las regiones resultantes se las nombra regiones de Voronoi o polígonos de Voronoi (denotado Vor(pi)). Los puntos del conjunto reciben el nombre de generadores del diagrama.

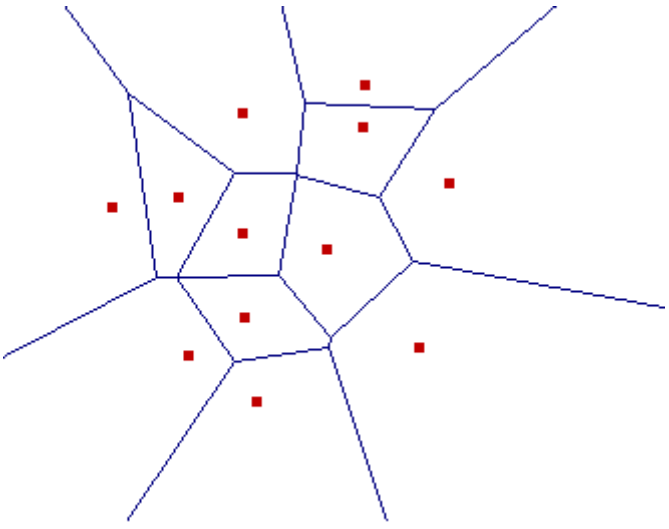


Figura 24. Diagrama de Voronoi

En la figura se muestra el diagrama de Voronoi de una nube puntos (en rojo) en el plano.

Este método tiene ciertos problemas cuando se trata de objetos delgados, además de discontinuidad en las fuerzas.

Otro método es mediante el uso de restricciones que consiste en colocar el PROXY en el punto más cercano al HIP que cumpla las restricciones impuestas. Por simplicidad se obliga a que el PROXY se coloque sobre el plano de restricción. Y se minimiza la distancia entre estos mediante el uso de multiplicadores de Lagrange.

$$A_i \cdot x + B_i \cdot y + C_i \cdot z + D_i = 0$$

$$E = \frac{1}{2} \cdot k \cdot \left((x - x_{HIP})^2 + (y - y_{HIP})^2 + (z - z_{HIP})^2 \right)$$

$$C = \frac{1}{2} \cdot k \cdot \left((x - x_{HIP})^2 + (y - y_{HIP})^2 + (z - z_{HIP})^2 \right) + \\ + \sum_{i=1}^{3} \lambda_i \cdot (A_i \cdot x + B_i \cdot y + C_i \cdot z + D_i)$$

Se resuelve entonces el siguiente sistema para un máximo de tres planos:

$$\begin{pmatrix} 1 & 0 & 0 & A_1 & A_2 & A_3 \\ 0 & 1 & 0 & B_1 & B_2 & B_3 \\ 0 & 0 & 1 & C_1 & C_2 & C_3 \\ A_1 & B_1 & C_1 & 0 & 0 & 0 \\ A_2 & B_2 & C_2 & 0 & 0 & 0 \\ A_3 & B_3 & C_3 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{pmatrix} = \begin{pmatrix} x_{HIP} \\ y_{HIP} \\ z_{HIP} \\ D_1 \\ D_2 \\ D_3 \end{pmatrix}$$

Se trata de un proceso iterativo:

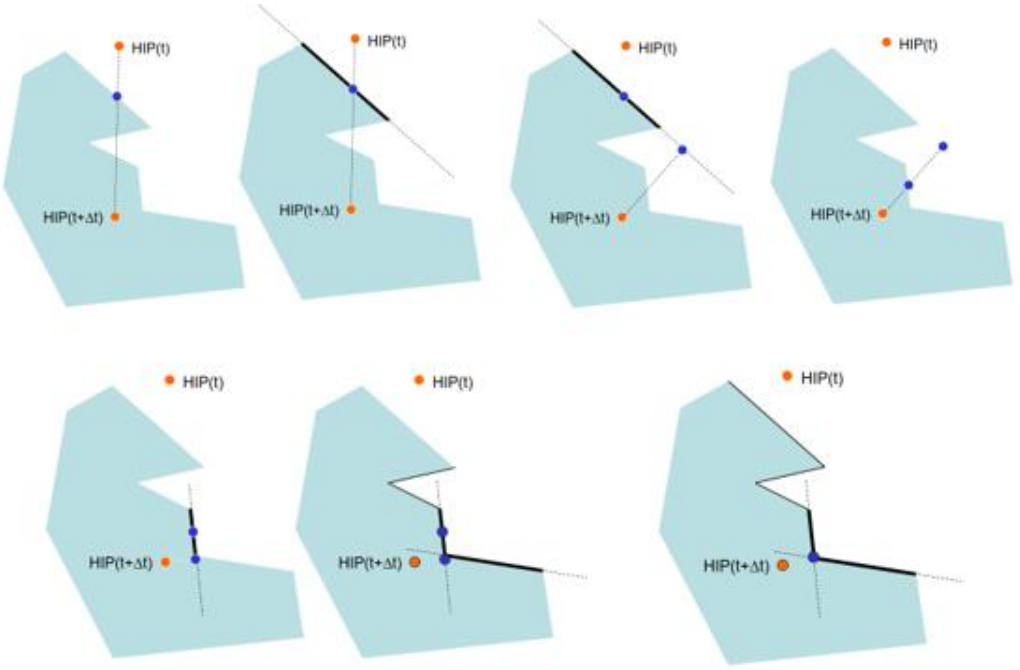


Figura 25. Proceso iterativo del cálculo del PROXY

Por último, está el método de utilizar una esfera en vez de un punto, esto aporta robustez al sistema eliminando huecos (errores de redondeo) y es fácilmente extensible cuando se modifica la geometría de los objetos. Es por ello que se empleó en este estudio este método debido a su sencillez y ventajas que conlleva.

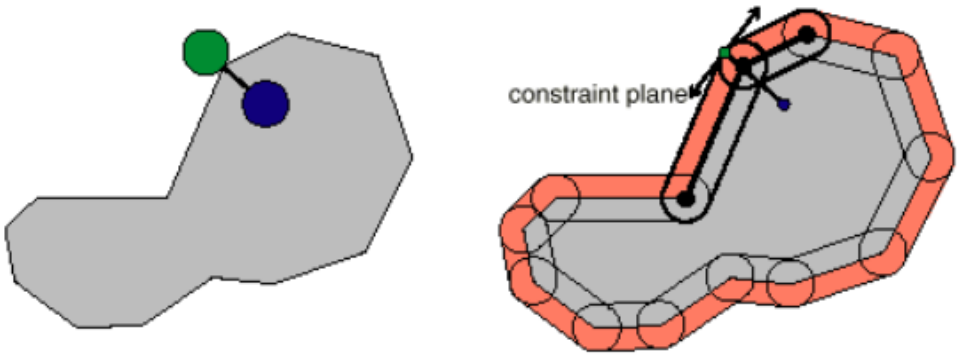


Figura 26. Proxy calculado con puntero esférico

Una vez conocidas la situación del PROXY y el HIP ya se estaría en disposición de aplicar la fuerza proporcional a la distancia entre ellos. Aunque en primer lugar es necesario que los objetos interactúen entre sí, es decir, tienen que poder colisionar. Es necesario un cálculo de la dinámica entre diferentes objetos.

2.6 CÁLCULO DE LA DINÁMICA MULTICUERPO

Para representar una tarea completa de ensamblaje virtual, es necesario representar con fidelidad el movimiento de las piezas en juego con respecto al paso del tiempo y de las actuaciones de los usuarios.

Las librerías del SDK de OpenHaptics no calculan otras fuerzas que se deriven del contacto directo con objetos inmóviles o con trayectorias predefinidas e inmutables. Casos tales como: objetos que se tocan, trozos de mecanismos que se puedan seguir moviendo aun cuando estén agarrados (como los trípodes o una cadena), máquinas completas...

Existen unas cuantas librerías que resuelven la dinámica de sistemas mecánicos, que se encuentran entre el ámbito científico y el del entretenimiento.

Para este proyecto se ha empleado una librería de código abierto destinada a simulaciones interactivas, Bullet. Está desarrollada principalmente por un antiguo empleado de Sony Computer Entertainment, aunque Bullet no es un proyecto de esta empresa. Bullet Physics está licenciado bajo la licencia zlib permisiva:

Este software se proporciona "tal cual", sin ninguna garantía explícita o implícita. En ningún caso los autores serán responsables de los daños y perjuicios derivados del uso de este software.

Se otorga permiso a cualquier persona que use este software para cualquier propósito, incluyendo aplicaciones comerciales, y para alterarlo y redistribuirlo libremente, sujeto a las siguientes restricciones:

1. El origen de este software no debe ser mal representado; No debe reclamar que escribió el software original. Si utiliza este software en un producto, se agradecerá un reconocimiento en la documentación del producto, pero no es necesario.

2. Las versiones de fuentes alteradas deben estar claramente marcadas como tales y no deben ser mal representadas como el software original.

3. Este aviso no puede ser removido o alterado de cualquier distribución de origen.

Por lo tanto, como se ha mencionado, es posible el uso de estas librerías para este proyecto y aplicaciones que consideremos.

Bullet proporciona:

- Simulación dinámica de objetos ante fuerzas aplicadas, como, por ejemplo: gravedad, fuerzas puntuales, de rozamiento, etc.

- Detección de contactos y cálculo de reacciones entre elementos.

- Imposición de restricciones entre piezas, como pares de revolución, esféricos, cilíndricos…

Debido a su trasfondo como librería de entornos interactivos y lúdicos, la simulación se desarrolla a una frecuencia típica de 60Hz.

2.6.1 Uso de la librería Bullet de dinámica de mecanismos

Bullet, como librería concebida para ser integrada en proyectos heterogéneos, no proporciona directamente funciones para todas las tareas externas que no sea la propia resolución del problema dinámico. En el caso de la representación 3D, el usuario es el encargado de representar por sí mismo todos los objetos de la escena en las posiciones correspondientes.

Para tal tarea se ha empleado la librería osgBullet, que permite visualizar directamente el estado de la simulación con la librería OpenSceneGraph (OSG). Además, osgBullet permite también desacoplar automáticamente las tareas del cálculo dinámico de Bullet y la representación gráfica Puesto que no tienen por qué tener las mimas exigencias en cuanto a tasa de refresco.

Bullet soporta estudio de dinámica de cuerpos rígidos y deformables, para este estudio solo se han empleado cuerpos rígidos. Los cuerpos rígidos tienen como principales características que, la distancia entre cualquier par de vértices que forman la malla nunca varía, que el campo de velocidades en función únicamente de la velocidad angular y lineal, la definición de masa a través de m, CDG e Ig.

El elemento principal en Bullet es el *mundo* (world). El mundo dentro de Bullet tiene varias responsabilidades, entre las que podemos destacar:

- Servir como estructura de datos donde almacenar los cuerpos físicos que lo conforman.

- Aplicar una serie de restricciones a estos cuerpos, como por ejemplo la de punto a punto o de bisagra, también se encarga de detectar y aplicar colisiones entre estos cuerpos y actualizar su posición automáticamente cuando se aplique cualquier tipo de fuerza sobre estos.

El mundo tiene diversas implementaciones dentro de la biblioteca, dependiendo de si utilizamos cuerpos rígidos o flexibles. En este caso se están usando cuerpos rígidos, de modo que la clase que se utilizará será btDiscreteDynamicsWorld.

Es importante describir el modo en el que se define el paso de tiempo que corre la simulación, debido a que dictará a que frecuencia se ejecuta. Esta tarea se lleva a cabo haciendo una llamada a la función StepSimulation() y pasándole como argumentos el paso de tiempo que se desea. Que en este caso consiste en restar al instante actual el instante previo del ciclo anterior.

Tras haber tenido en cuenta todos estos aspectos podría comenzarse con la creación de cuerpos.

2.6.1.1 Creación de cuerpos

Los cuerpos y formas de colisión pueden crearse una vez hayan sido inicializada la biblioteca. El método para la creación de cuerpos se explica a continuación:

El constructor de la clase btRigidBody recibe un objeto btRigidBody::btRigidBodyConstructionInfo. Este objeto sirve para inyectar al constructor de la clase información relativa al cuerpo rígido que se va a crear. Los argumentos que recibe son la masa del objeto, el estado del cuerpo (Motion State del cuál hablaremos más adelante), la definición geométrica del cuerpo(btCollisionShape) y el tensor de inercia. Cabe mencionar que Bullet asigna como origen de coordenadas y centro de gravedad el mismo punto de origen que el que hayamos puesto como origen en el programa de modelado.

La librería comprende tanto objetos móviles como inmóviles, según la asignación de masa que se les otorgue. Así, un objeto con masa nula se considera como objeto inmóvil. Esto podría ser útil para la creación de un suelo o un entorno en el que se apoyen o con el que colisiones los objetos móviles.

A partir de la forma del cuerpo y de la masa de este, Bullet calcula el tensor de inercia del cuerpo físico a partir de la distribución de vértices, aunque es posible asignársela de forma manual si ya es conocida. Sin embargo, esto se trata de una aproximación y no representa fielmente la realidad, además del hecho de que para formas complejas las aproximaría a demasiado simples.

Una vez creado el cuerpo, hay que añadirlo a la simulación mediante la notificación de Bullet correspondiente.

Bullet ofrece una gran variedad de formas primitivas de colisión, entre las que podemos listar:

- btBoxShape: caja definida por el tamaño de sus lados.
- btSphereShape: esfera definida por su radio.
- btCapsuleShape: capsula alrededor del eje Y. También existen btCapsuleShapeX/Z.
- btCylinderShape.
- btConeShape: cono alrededor del eje Y. También existen btConeShapeX/Z.
- btMultiSphereShap: cascarón convexo formado a partir de varias esferas que puede ser usado para crear una cápsula (a partir de dos esferas) u otras formas convexas.

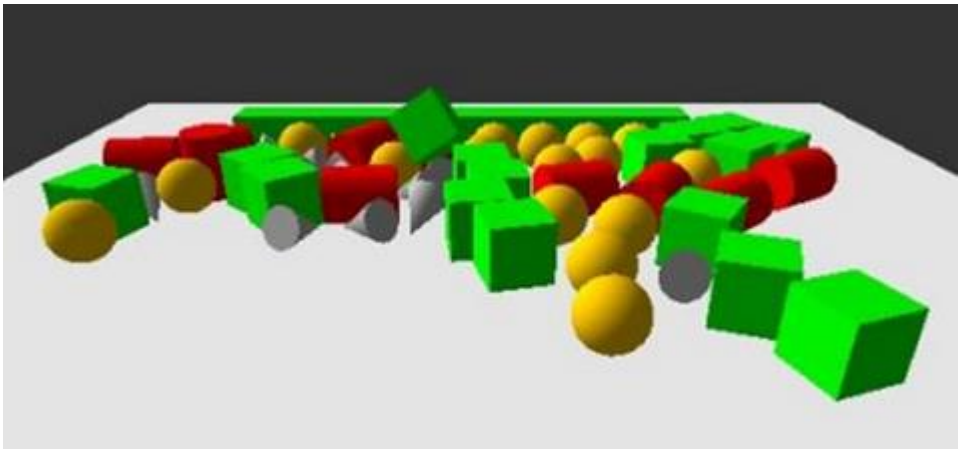


Figura 27. Variedad de formas primitivas de colisión

Cada una de estas especializaciones puede ser beneficiosa dependiendo de la naturaleza exacta de la geometría, cuanto más simple sea la forma tendrá menos coste computacional y en el caso de que no sea ajuste a ninguna forma sencilla, las únicas alternativas serían: el contacto de malla genérico (que es más costoso y flexible) o bien la composición de formas de colisión complejas a partir de primitivas simples.

Bullet tambien ofrece formas compuestas, pudiendo combinar múltiples formas convexas en una única usando la clase btCompoundShape. Esto es deseable para una mayor fidelidad de las formas, pero a cambio de un mayor coste computacional. Cada una de las formas que dan lugar a la malla principal se llama forma hija. Cada forma hija tiene sus propias transformaciones locales, relativas a la forma compuesta. Existen algunas formas de colisión más avanzadas que permiten ajustarse a geometrías que no corresponden con formas primitivas.

En la figura siguiente se muestra una ayuda en la elección de la forma de colisión adecuada dependiendo de la situación.

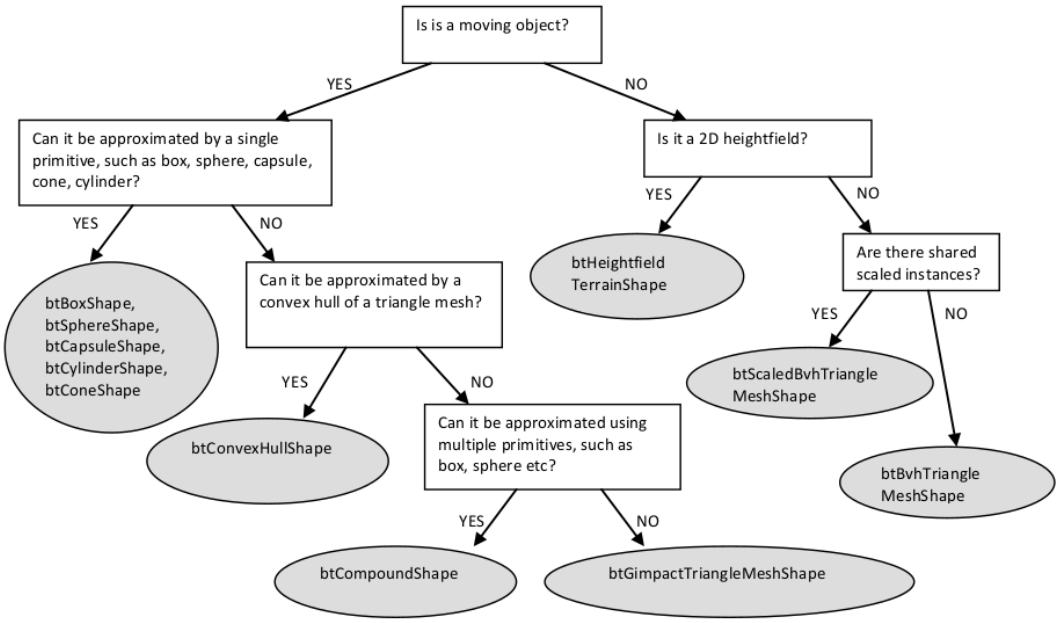


Figura 28. Esquema de elección de la forma de colisión

Para construir un cuerpo rígido, se define una transformación a partir de un cuaternio y un vector, el cuaternio unitario proporciona una notación matemática para representar las orientaciones y las rotaciones de objetos en tres dimensiones. Los cuaternios generalmente son representados de la siguiente forma:

$$a + bi + cj + dk$$

Donde a, b, c y d son números reales y i,j y k son las unidades fundamentales del cuaternio.

Comparados con los ángulos de Euler (α, β, γ,) son más simples de componer y evitan el problema del bloqueo del cardán, que consiste en la pérdida de un grado de libertad en una suspensión cardán de tres rotores, ocurre cuando los ejes de dos de los tres rotores se colocan en paralelo, bloqueando el sistema en una rotación en un espacio bidimensional degenerado.

Los cuaternios comparados con las matrices de rotación, son más eficientes y estables numéricamente.

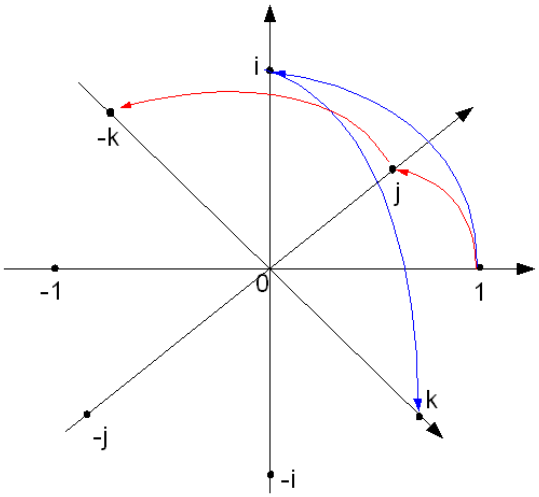


Figura 29. Representación gráfica de los cuaternios unitarios

El cuaternio recibe un vector, mediante el cual le indicamos en qué ejes sufrirá rotaciones nuestro objeto, y un parámetro entre 0 y 2π con el que indicamos los radianes de rotación.

El segundo parámetro de la transformación es el vector de traslación, que indica el lugar donde se creará el objeto con respecto al sistema de referencia global de la simulación.

Una vez creado el cuerpo rígido, se puede definir los parámetros correspondientes al modelo de contacto entre sólidos: el coeficiente de restitución, que indica una medida del grado de conservación de la energía cinética tras el choque entre un par de objetos, y el coeficiente de fricción, que indica la oposición al deslizamiento que ofrecen dos superficies en contacto.

2.6.1.2 Detección de colisiones

La detección de contactos genérica entre objetos es un problema computacionalmente exigente debido al elevado número de posibles contactos entre los objetos de la escena y el cálculo exacto de los puntos de colisión. El problema se simplifica dividiéndolo en dos etapas:

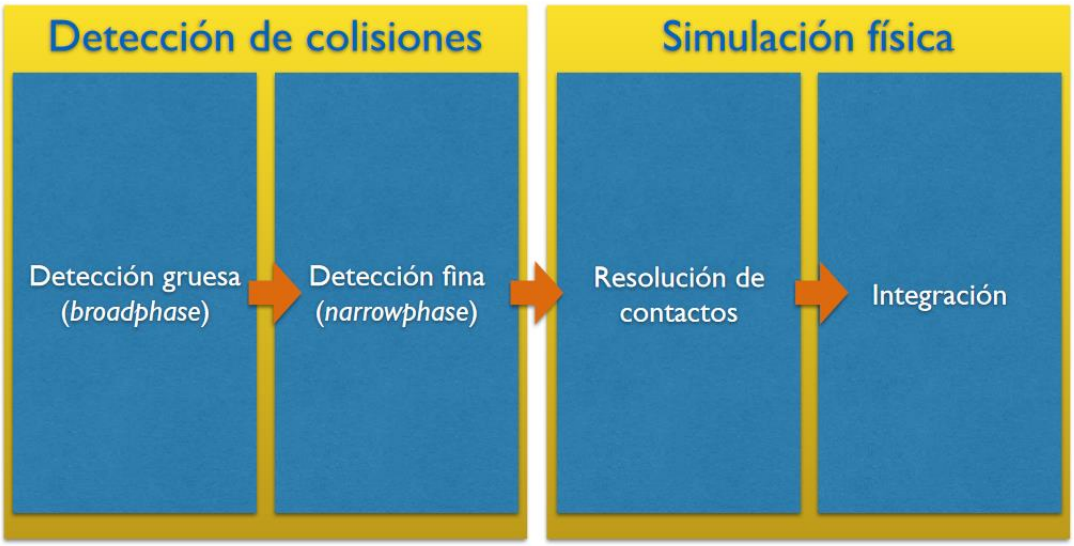


Figura 30. Orden lógico de la detección de colisiones

1. El problema combinatorio se resuelve en la fase ´´amplia`` o BroadPhase que segmenta el espacio de simulación, y encuentra las parejas de objetos potencialmente en contacto. Esta fase emplea grandes simplificaciones de la geometría, encerrándolas en primitivas sencillas como cilindros, esferas conos, etc.

2. El problema de detalle se resuelve en la fase ´´reducida`` o NarrowPhase, donde se calculan los puntos exactos y las posibles interpenetraciones entre objetos candidatos de la etapa anterior.

Bullet se encarga de resolver las ecuaciones de la dinámica y de satisfacer simultáneamente las restricciones.

Sin embargo, existen casos en los que se desea diferenciar entre colisiones, con el fin de que unos determinados objetos sólo pueden colisionar con otros determinados. Es entonces cuando surge la necesidad de un filtrado de colisiones que se explicará a continuación.

La fase de detección de colisiones, hace uso de una serie de estructuras de datos:

- btCollisionObject: almacena las formas de colisión y las transformaciones de éstas.

- btCollisionShape: describe la forma de colisión de un objeto de colisión, tal como una caja, una esfera, una forma convexa (convex hull) o una malla de triángulos. Una forma de colisión puede ser compartida entre múltiples objetos de colisión.

- btCollisionWorld: almacena todos los objetos de colisión y proporciona una interfaz que permite realizar peticiones de forma eficiente. Nuestro objeto

dynamics_world_ es de tipo btDiscreteDynamicsWorld, que es una subclase de ésta.

2.6.1.2.1 Filtrado de colisiones

Bullet proporciona tres maneras de asegurar que sólo ciertos objetos chocan entre sí: máscaras, devoluciones de llamada de filtro de fase amplia y nearcallbacks personalizados.

Vale la pena señalar que la selección de colisión basada en máscara ocurre mucho más arriba en la cadena de herramientas que la devolución de llamada. En resumen, las máscaras funcionan mejor y son mucho más fáciles de usar.

- Uso de máscaras: Bullet soporta máscaras bit a bit como una forma de decidir si las cosas deben o no chocar con otras cosas, o recibir colisiones. Por ejemplo, en un juego de nave espacial, se podría hacer que las naves espaciales ignoren las colisiones con otras naves espaciales, pero siempre chocan con las paredes. Para ello se le pasará como argumento, al crear el cuerpo, la máscara que tiene asignada el objeto. Antes de buscar una colisión, bullet realiza un AND entre la máscara A del objeto y el grupo B del objeto. Si el resultado es cualquier cosa menos 0, Bullet realizará la respuesta de colisión.

- Filtrado de colisiones mediante una devolución de llamada de filtro de fase amplia: es una forma eficiente de filtrado, consiste en registrar una devolución de llamada de filtro de fase ancha. Esta devolución de llamada se llama en una etapa muy temprana, y evita la generación de pares de colisión.

- Filtrado de colisiones utilizando un NearCallback personalizado: se puede registrar otra devolución de llamada durante la NarrowPhase, cuando todos los pares son generados por la BroadPhase.

Profundizando más en detalle sobre las máscaras, ya que es el método recomendado y el usado en este proyecto, es necesaria una aclaración de cómo se comportan y como compararlas.

Las máscaras son básicamente un vector de bits que se comparan con las llamadas flags o ´´banderas``, que son también bits que se le asignan como una identidad específica para cada objeto.

Una vector de bits es una asignación de algún dominio (casi siempre un rango de números enteros) a valores en el conjunto {0, 1}. Los valores se pueden interpretar como oscuro / claro, ausente / presente, bloqueado / desbloqueado, válido / no válido, etcétera. El punto es que sólo hay dos valores posibles, por lo que se pueden almacenar en un bit. Al igual que con otros vectores, el acceso a un solo bit se puede administrar mediante la aplicación de un índice al vector. Suponiendo que su tamaño (o longitud) sea n bits, la matriz se puede utilizar para especificar un subconjunto del dominio (por ejemplo, {0, 1, 2, ..., n -1}), donde un 1-bit indica la presencia y un 0-bit la ausencia de un número en el conjunto.

Aunque la mayoría procesadores no son capaces de direccionar bits individuales en la memoria, ni tienen instrucciones para manipular bits individuales, cada bit puede ser seleccionado y manipulado usando operaciones bit a bit. En particular:

- OR se puede utilizar para establecer un bit a uno:
  11101010 OR 00000100 = 11101110
- AND se puede utilizar para establecer un bit a cero:
  11101010 AND 11111101 = 11101000
- AND junto con la prueba de cero se puede utilizar para determinar si un bit está establecido:
  11101010 AND 00000001 = 00000000 = 0
  11101010 AND 00000010 = 00000010 ≠ 0
- XOR se puede utilizar para invertir o alternar un poco:
  11101010 XOR 00000100 = 11101110
  11101110 XOR 00000100 = 11101010
- NOT se puede utilizar para invertir todos los bits:
  NOT 10110010 = 01001101

Para obtener la máscara de bits necesaria para estas operaciones, podemos utilizar un operador de cambio de bit para cambiar el número 1 a la izquierda por el número de lugares apropiado, así como la negación en el bit si es necesario.

Se puede explicar todo este método a partir de un ejemplo, que es precisamente el implementado en este proyecto: se trata de definir unas flags de colisión para las esferas de los punteros y otras para el resto de objetos.

Col_Pointer=0x1<<1; -> 0010

Col_Default=0x1<<2; -> 0100

Aquí se ha creado una flag para cada tipo de objeto, que posteriormente se pasará como argumento cuando se cree dicho objeto. Asignándole un vector de bits a las esferas de los punteros y otro vector de bits distinta para el resto de objetos.

Seguidamente se definen las máscaras para establecer que objetos pueden colisionar con que otros.

Pointer_Collides_With=Col_Default -> 0100

Default_Collides_With=Col_Default|Col_Pointer -> 0110

Ahora mediante una operación OR se puede comparar que flag coincide con que máscara.

2.6.1.3 Información de contactos

A partir de la detección de colisiones explicada anteriormente, es posible obtener información acerca de los contactos que se dan entre los objetos de la escena.

Esta información la proporciona el gestor de colisiones btCollisionDispatcher, que agrupa pares de objetos en colisión, puede devolver información relevante como: localización en coordenadas locales del punto de contacto o profundidad de penetración entre los objetos.

Sin embargo, no es posible detectar en eses pares de colisión, la información obtenida a que objeto de eses dos hace referencia. Entonces se recurre al uso de IDs.

El uso de IDs consiste en un método similar a las flags y explicadas anteriormente. Se le asigna un número entero a cada objeto, que será su identificador. Posteriormente en el momento que sea necesario, mediante una llamada se puede obtener el identificador del objeto con el que se esté tratando. De este modo es posible conocer que objetos están colisionando y cual es cual.

2.6.1.4 Restricciones

Las restricciones limitan el movimiento de dos cuerpos rígidos en relación entre sí, o el movimiento de un cuerpo en relación con el espacio global. Bullet ofrece 5 tipos de restricciones:

1. Restricción de la bisagra: restringe el movimiento de dos cuerpos por medio de un eje compartido. El eje está definido por un punto de pivote en cada cuerpo (dentro del espacio local del cuerpo). Las limitaciones de las bisagras se pueden utilizar, por ejemplo, para modelar puertas.

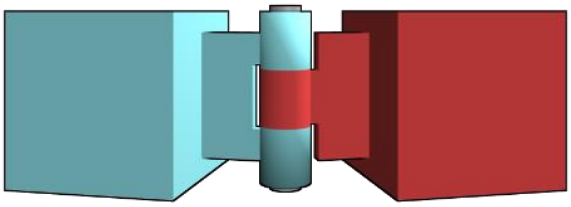


Figura 31. Ejemplo de restricción de visagra

2. Restricción del deslizador: permite que los dos cuerpos se muevan a lo largo de un eje compartido. La rotación alrededor del pistón puede ser limitada, si es necesario.

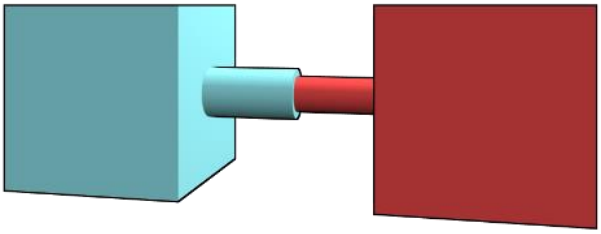


Figura 32. Ejemplo de restricción de deslizador

3. Restricción Esférica: modela una conexión de bola entre dos cuerpos rígidos.

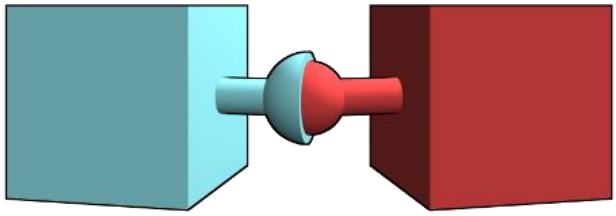


Figura 33. Ejemplo de restricción de tipo esférica

4. Restricción de cono: es una versión especializada de la restricción esférica. Permite limitar la rotación y el balanceo (en ambas direcciones perpendiculares).

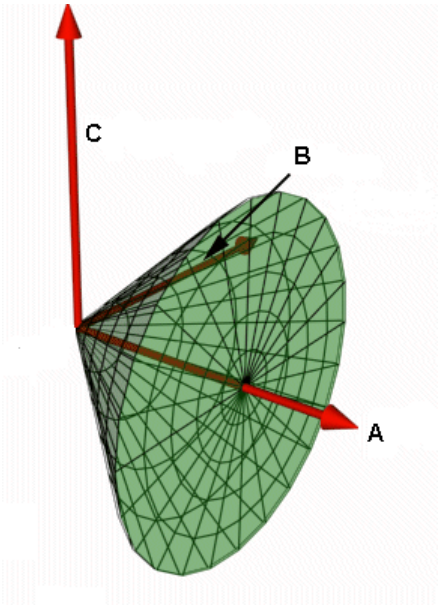


Figura 34. Restricción de tipo cono

5. Restricción genérica: permite el movimiento en todos los seis grados de libertad, y permite limitar este movimiento como se desee. Los seis grados de libertad (6DF) todos los movimientos que puede hacer un cuerpo rígido en el espacio tridimensional. Específicamente, el cuerpo es libre de cambiar de posición como avance / retroceso (subida), arriba / abajo (impulso), izquierda / derecha (balanceo) en tres ejes perpendiculares, combinado con los cambios de orientación a través de la rotación de tres ejes perpendiculares, guiñada (eje normal), cabeceo (eje lateral) y balanceo (eje longitudinal).

2.6.1.4.1 Parámetros de distinción de restricciones

Existen ocasiones en que las restricciones entran en conflicto y es necesario priorizar unas frente a otras, para ello se emplean los parámetros ERP y CFM. Estos parámetros controlan la distinción entre restricciones duras y flexibles.

2.6.1.4.1.1 ERP

El error de articulación y el parámetro de reducción de errores ERP: Cuando una unión conecta dos cuerpos, se requiere que dichos cuerpos tengan posiciones y orientaciones relativas entre sí determinadas. Sin embargo, es posible que los cuerpos se encuentren en posiciones que no se cumplan las restricciones conjuntas. Este "error conjunto" puede ocurrir por dos motivos:

1. Si el usuario establece la posición / orientación de un cuerpo sin ajustar correctamente la posición / orientación del otro cuerpo.
2. Durante la simulación, los errores pueden influir haciendo que los cuerpos se alejen de sus posiciones requeridas.

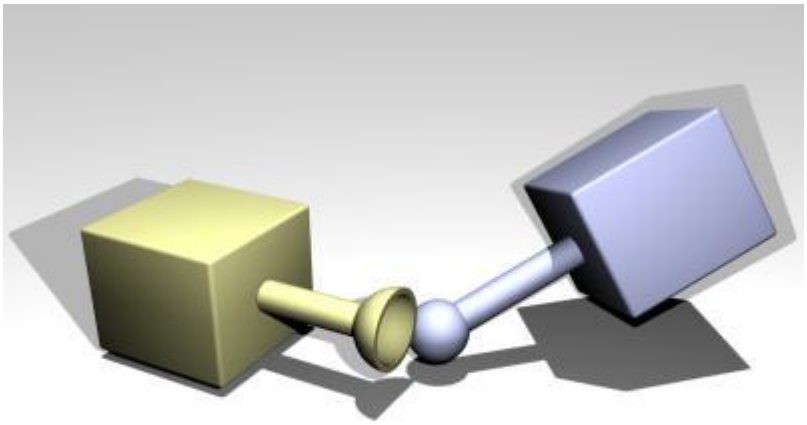


Figura 35. Situación de rotura de una restricción

Hay un mecanismo para reducir el error de la junta: durante cada paso de simulación cada junta aplica una fuerza especial para traer sus cuerpos de nuevo en la alineación correcta. Esta fuerza es controlada por el parámetro de reducción de errores (ERP), que tiene un valor entre 0 y 1.

El ERP especifica qué proporción del error de la articulación se fijará durante el próximo paso de simulación. Si ERP = 0 entonces no se aplica ninguna fuerza de corrección y los cuerpos finalmente se alejarán a medida que avanza la simulación. Si ERP = 1, entonces la simulación intentará corregir todos los errores de las juntas durante el siguiente paso de tiempo. Sin embargo, el ajuste de ERP = 1 no es recomendable, ya

que el error de articulación no será completamente fijo debido a varias aproximaciones internas. Se recomienda un valor de ERP = 0.1 a 0.8 (0.2 es el valor predeterminado).

Se puede establecer un valor ERP global que afecta a la mayoría de las articulaciones de la simulación. Sin embargo, algunas articulaciones tienen valores ERP locales que controlan varios aspectos de la articulación.

2.6.1.4.1.2 CFM

La mayoría de las restricciones son por naturaleza "duras". Esto significa que las restricciones representan condiciones que nunca son violadas. Por ejemplo, la bola debe estar siempre en el zócalo, y las dos partes de la bisagra siempre deben estar alineadas. En la práctica, las restricciones pueden ser violadas por la introducción involuntaria de errores en el sistema, pero el parámetro de reducción de errores se puede configurar para corregir estos errores.

Para ello se emplea el de mezcla de fuerzas de restricción CFM. Si el CFM se pone a cero, la restricción será dura. Si CFM se establece en un valor positivo, será posible violar la restricción "empujándola" (por ejemplo, para las restricciones de contacto forzando a los dos objetos en contacto juntos). En otras palabras, la restricción será flexible, y la flexibilidad aumentará a medida que aumenta CFM.

No todas las restricciones son inviolables, algunas restricciones "flexibles" están diseñadas para no serlo. Por ejemplo, la restricción de contacto que impide que los objetos que chocan penetre es dura por defecto, por lo que actúa como si las superficies de colisión fueran indeformables. Pero puede convertirse en una restricción flexible para simular materiales más blandos, permitiendo así una cierta penetración natural de los dos objetos cuando se ejerce una fuerza que los junte.

Lo que realmente sucede aquí es que la restricción permite ser violada por una cantidad proporcional a CFM veces la fuerza de restauración que se necesita para hacer cumplir la restricción. Hay que tener en cuenta que la configuración de CFM a un valor negativo puede tener efectos negativos no deseados, como la inestabilidad.

2.7 VISUALIZACIÓN DE LOS MODELOS 3D

Un grafo de escena consiste en un grafo acíclico dirigido (DAG). Comienza con un nodo raíz, que contiene todo el mundo virtual, ya sea 2D o 3D. El mundo se distribuye en una jerarquía de nodos que representan las agrupaciones espaciales, los ajustes de la posición, las animaciones de los objetos o las definiciones de las relaciones lógicas entre ellos.

Las hojas del grafo representan los objetos físicos, es decir, las geometrías dibujables en caso de una escena tridimensional y sus características materiales. Un nodo del grafo puede tener varios padres, como, por ejemplo, el caso de las ruedas en los coches o los caballetes de esta simulación.

Un grafo no es el motor completo que ejecuta un videojuego o una simulación, tan solo representa la escena a visualizar del mismo y puede formar una parte importante de dichos sistemas. Es una estructura de datos independiente y que puede servir para muchos fines distintos.

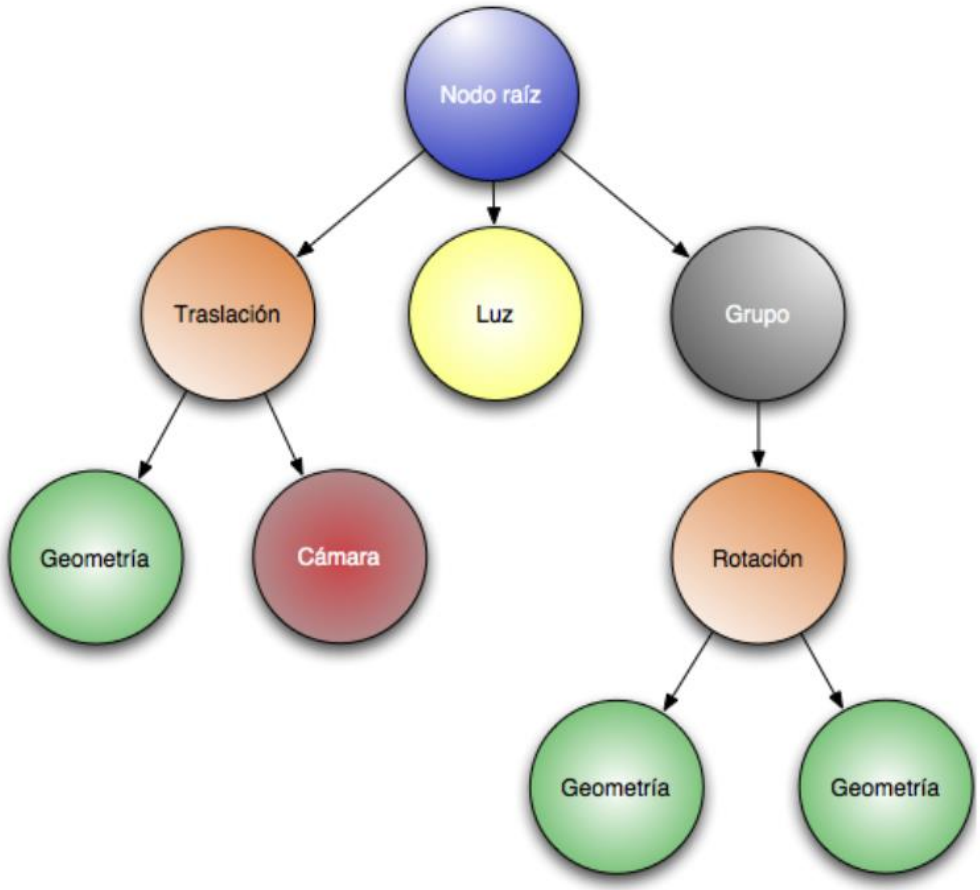


Figura 36. Nodos de un grafo

2.7.1 Beneficios del uso de un grafo de escena

Las razones clave por las que en muchos desarrollos gráficos se hace uso de un grafo de escena son la eficiencia, productividad, portabilidad y escalabilidad:

- Eficiencia: Los grafos de escena proveen un excelente entorno de trabajo para aumentar la eficiencia. Un buen grafo de escena emplea dos técnicas clave. Cálculo de la visibilidad para objetos que no son visibles, y un estado ordenado de propiedades como son texturas, materiales de los objetos, luces, transformaciones y cámaras.

- Productividad: Quita mucho trabajo en el desarrollo de aplicaciones gráficas de alto rendimiento. Uno de los conceptos más importantes en la programación orientada a objetos es la composición de objetos (objetos que contienen otros objetos), lo cual hace que sea un diseño altamente flexible y reusable. En pocas palabras permite adaptarse fácilmente para resolver los problemas relacionados con este tema.

- Portabilidad: Abstrae los detalles de las tareas de bajo nivel, permitiendo la exportación entre múltiples plataformas.

- Escalabilidad: Junto con el hecho de que los grafos de escena pueden manejar escenas de gran complejidad, también facilitan la tarea de manejar complejas configuraciones hardware, como son clusters, o sistemas multiprocesador.

2.7.2 OpenSceneGraph

OpenSceneGraph es una herramienta de software libre para el desarrollo de aplicaciones gráficas de alto rendimiento como son simuladores de vuelo, juegos, realidad virtual y visualización científica. Basado en el concepto de grafo de escena, provee un entorno de trabajo orientado a objetos por encima de OpenGL (librería estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D).

El propósito de OpenSceneGraph es el uso de la tecnología del grafo de escena tanto para propósitos comerciales como no comerciales. Se encuentra escrito completamente en C++ y OpenGL, haciendo uso intensivo de la STL y de patrones de diseño.

Los puntos clave de OpenSceneGraph son su rendimiento, escalabilidad, portabilidad y la productividad todas asociadas al hecho de usar un completo grafo de escena. OpenSceneGraph está dividido en varios componentes, de los cuales el único básico es osg core, el resto añaden funcionalidad externa, las más relevantes se muestran a continuación:

- Formatos de fichero: Carga de escenas y manejo de datos.

- osgUtil: Funciones de utilidad para el procesado y modificación de la escena y cálculo de la visibilidad.

- osgViewer: Librería para la inicialización y manejo de ventanas gráficas de manera independiente de la plataforma.

OpenSceneGraph libera al desarrollador de la implementación y optimización de llamadas gráficas de bajo nivel, además de añadir multitud de utilidades para el rápido desarrollo de aplicaciones gráficas.

Mediante el empleo de OpenGL se definen las geometrías y se muestra en la pantalla del ordenador el espacio virtual. Este espacio es en realidad una proyección de un frustrum tridimensional, que es una porción de una figura geométrica (usualmente un cono o una pirámide) comprendida entre dos planos paralelos.

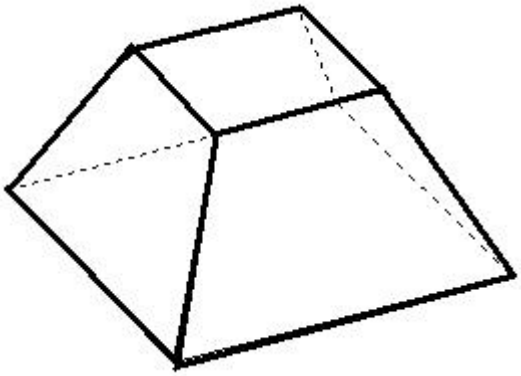


Figura 37. Frustrum

La tercera dimensión se proyecta a cierta profundidad por debajo de la superficie de la pantalla. Este espacio tridimensional es el espacio mundial, puede ser descrito y dirigido por un sistema de coordenadas.

2.7.2.1 Cámara

Las escenas en un entorno 3D se representan siempre desde un punto, este punto se le denomina cámara.

La cámara establece el punto de vista (eyepoint, en el lenguaje de OpenGL) en el espacio virtual, es decir, la cámara define una vista dentro de este.

Es recomendable cambiar la cámara a medida que se desplace el punto de vista, en vez de girar toda la escena. Esto requeriría un mayor coste computacional.

La ubicación predeterminada para la cámara se muestra en el siguiente diagrama y se determina de la siguiente manera:

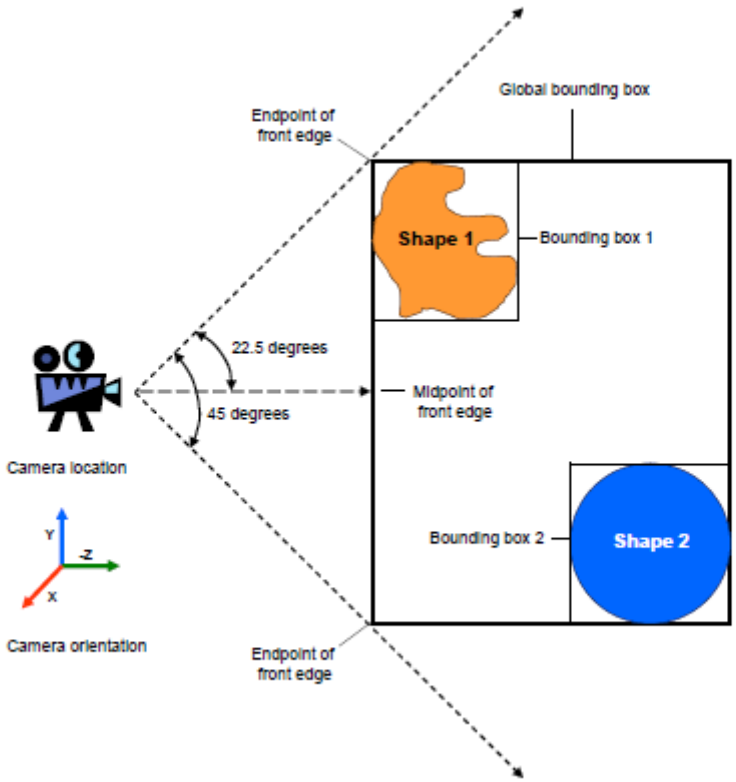


Figura 38. Localización de la cámara por defecto

Cada forma que se define dentro del espacio mundial tiene una caja que la delimita de forma implícita (consiste en un prisma rectangular) que abarca la forma. Todas estas cajas, cuando se combinan, comprenden una única global que contiene todos los objetos dentro de la misma.

De forma predeterminada, la ubicación de la cámara se coloca de forma que se mantiene un ángulo de 22,5 grados entre cada extremo del borde frontal de la caja delimitadora y una línea imaginaria dibujada de la cámara al punto medio del borde.

La dirección de visualización predeterminada para la cámara está a lo largo del eje -Z (eje Z negativo).

El espacio mundial tiene dos planos delimitadores:

- El plano de recorte frontal establece donde la cámara empieza a ver el espacio, ningún objeto más cercano a la cámara no será visible.

- El plano de recorte trasero fija el límite trasero de la cámara cualquier forma más lejana que ese plano no será visible.

Los planos de recorte predeterminados se muestran en el siguiente diagrama y se calculan de la siguiente manera:

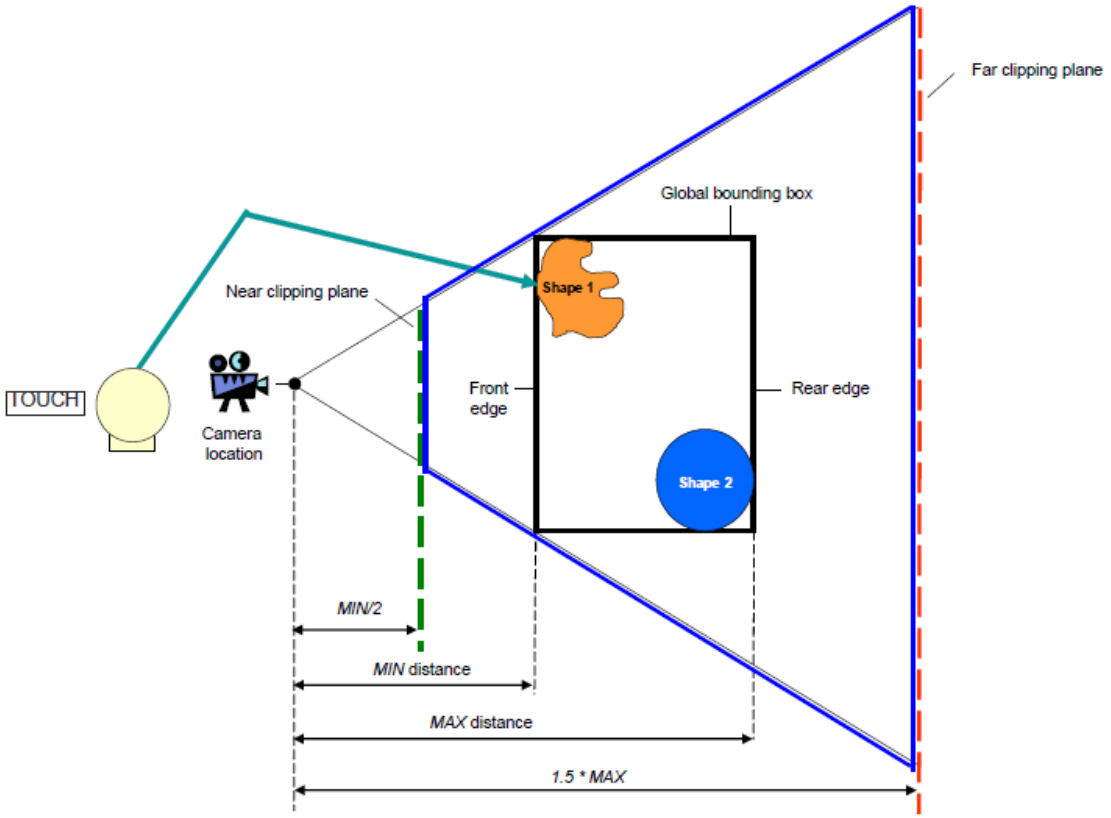


Figura 39. Localización de los planos de recorte

El plano de recorte frontal se encuentra a la mitad de la distancia desde la ubicación de la cámara hasta el borde frontal de la caja delimitadora global y el plano de recorte trasero se encuentra a 1,5 veces la distancia desde la ubicación de la cámara hasta el borde trasero de la caja delimitadora.

Obvia decir que la ubicación de la cámara no afecta directamente al espacio virtual.

3. IMPLEMENTACIÓN

Para este estudio se optó por realizar una simulación que consista en un montaje simple: recoger dos caballetes del suelo, abrirlos y colocarlos en posición para soportar una pieza de chapa metálica encima.

Al comienzo de este proyecto se pretendía usar simplemente las librerías proporcionadas por los dispositivos (OpenHaptics) para la parte de simulación y comunicación entre los hápticos y el ordenandor, sin embargo, se rechazó la opción del análisis de colisiones con esta librería a medida que se avanzó en este. Se tratará cronológicamente la problemática que fue surgiendo.

3.1 DEFINICIÓN DE GEOMETRÍA DE SÓLIDOS

En el manual de QuickHaptics menciona que admite la carga de objetos en formato de malla triangular, que consiste en la aproximación de una superficie 3D por una colección de triángulos y vértices.

Los paquetes de CAD como SolidWorks trabajan con geometría limitada por superficies, y permite una exportación de la pieza en formato .stl, sin embargo, QuickHaptics no admite tal formato. Es necesario un paso previo con otro programa, en nuestro caso empleamos Blender, el cual permite la conversión de .stl a .obj que sí que es admitido por QuickHaptics.

Se puede comprobar que un mayor número de triángulos para nuestras mallas (Trimesh) en Blender proporciona mayor precisión en el contacto de la pieza, pero un comportamiento más lento del programa, siendo incluso en casos extremos imposible iniciarlo. En cambio, un mallado pobre repercute en un comportamiento del puntero como si la superficie tuviera grietas. Para este estudio se opta entonces un número de triángulos en el mallado por defecto.

Cabe mencionar la coherencia necesaria en las orientaciones de los ejes y orígenes de coordenadas de las piezas cuando se exporta de un programa a otro, además del escalado (en nuestro caso fue de 10:1 respecto a la realidad).

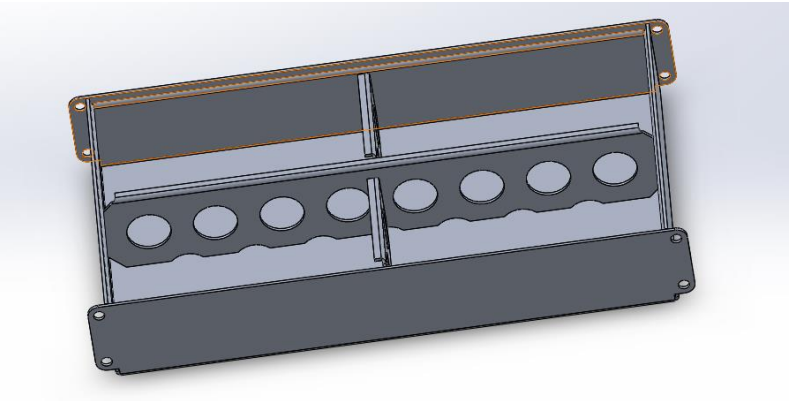


Figura 40. Pieza de chapa metálica diseñada en SolidWorks

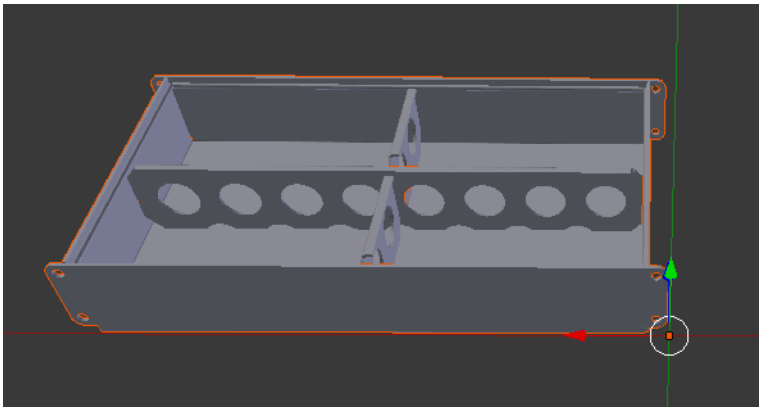


Figura 41. Pieza de chapa metálica importada a Blender

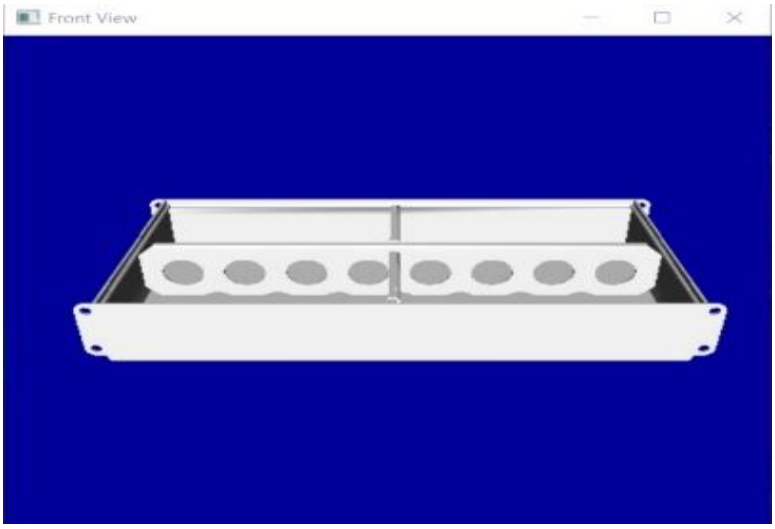


Figura 42. Pieza de chapa metálica importada a OpenHaptics

Este mallado tiene especial importancia para la creación de un cascarón ´´HULL`` que dará la forma a la pieza y proporciona al detector de colisiones los datos necesarios para calcular unas respuestas lógicas a las colisiones. A raíz de esto comienza a ser necesaria una librería más completa que Open Haptics en lo referente a colisiones, como, por ejemplo, Bullet. El estudio de colisiones de Open Haptics es mucho más simple y no contempla la situación de contactos entre objetos de la escena, sino simplemente la interacción entre un objeto y el puntero. En la siguiente figura se puede apreciar como dos objetos pueden compartir un mismo espacio, es decir, se atraviesan.

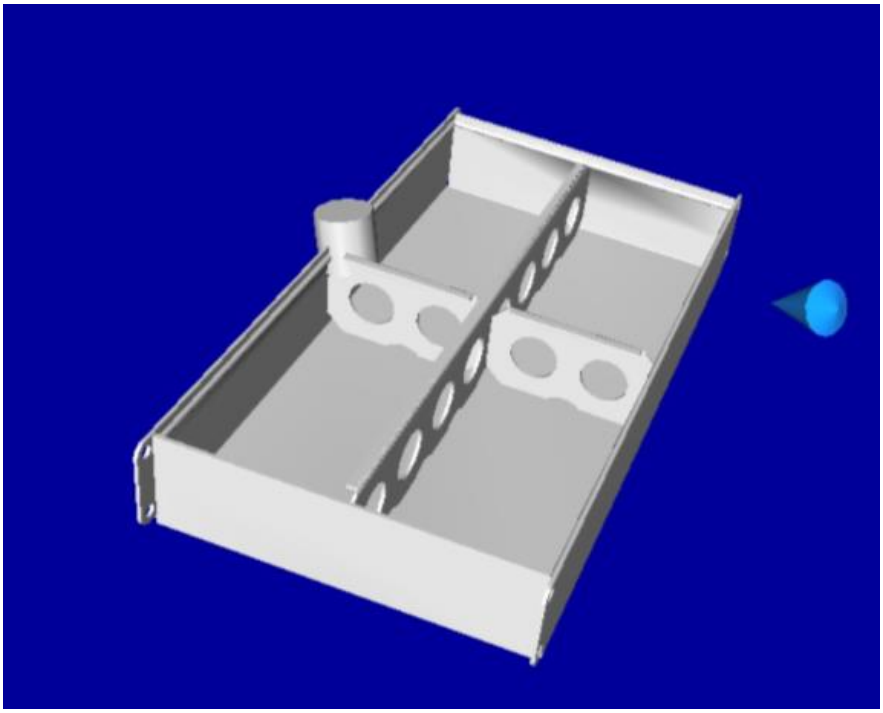


Figura 43. Situación indeseada de OpenHaptics en la que los objetos no interactúan entre si

Para la importación de sólidos en Bullet a partir de una pieza diseñada con SolidWorks, sigue siendo necesaria una transformación previa con Blender que pueda exportar estos archivos a formato.obj. La importación en Bullet se lleva a cabo en el momento de creación de los cuerpos (tema tratado anteriormente), tras crear un nodo en Open Scene Graph se hace la llamada a la función:

```
osgDB::readNodeFile("chapa.obj");
```

Que importará en este caso la geometría de la pieza ´´chapa``. Una vez creado el sólido se pueden realizar rotaciones y translaciones de los objetos en la inicialización de la escena mediante las funciones osg::Matrix::rotate() y osg::Matrix::translate(), que pasan a Open Scena Graph matrices de rotación y traslación para aplicar a cada objeto.

En el entorno de Bullet el escenario quedaría de la siguiente forma, donde los objetos pueden, ahora sí, interactuar:

Figura 44. Montaje final en entorno de Bullet con interacción correcta entre objetos en la escena

Finalmente aplicando texturas, colores, luces y sombras (mencionar que la tarjeta gráfica del ordenador empleado en este estudio no fue capaz de realizar dichas sombras). Open Scene Graph proporciona funciones que facilitan la aplicación de todas ellas, (osgShadow::ShadowedScene, settexture(), setlight(),etc).

El montaje de la simulación quedaría de la siguiente forma:

Figura 45. Montaje final

En cuanto se comenzó a interactuar con los objetos de la escena surgió la necesidad de implementar el manejo de los punteros mediante los dispositivos hápticos. Para ello se trataron temas como el estudio de la rigidez, la definición de los cursores, la detección de colisiones y el agarre de objetos.

Todos ellos se tratan a continuación con más detalle.

3.2 ESTUDIO DE LA RIGIDEZ

En el proceso de modificar la rigidez de los objetos empleando el parámetro stiffnes o ´´rigidez`` que proporciona Open Haptics para aplicar a cada objeto, surge la duda de qué relación existe entre esa rigidez, que solo permite valores comprendidos entre 0 y 1, y la K real que nos ejerce el dispositivo. Teniendo en cuenta la ley de Hooke, que dice que la fuerza ejercida por un resorte es directamente proporcional a una constante K y el desplazamiento, cabe pensar que analizando las deformaciones y fuerzas ejercidas por el dispositivo se puede conocer la K real.

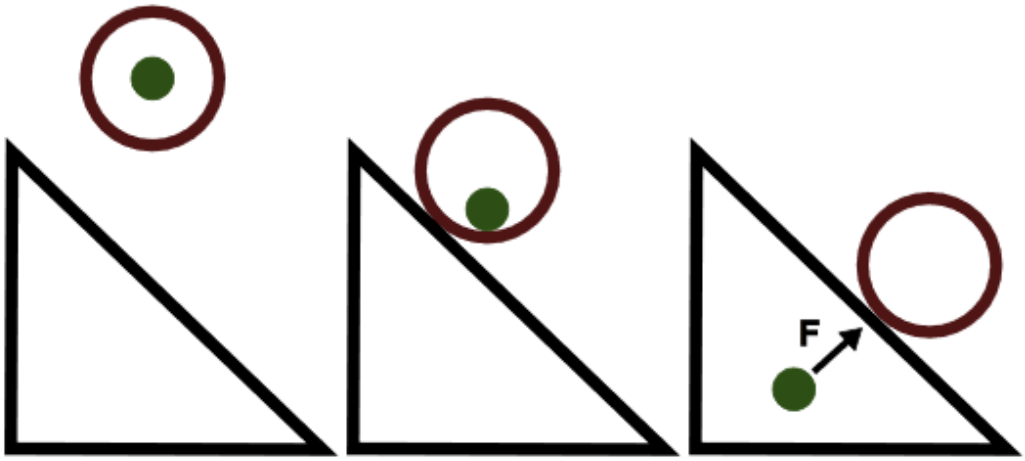


Figura 46. Situación de penetración del puntero ante un contacto

Teniendo en cuenta que los desplazamientos del dispositivo, con el uso de Quick Haptics, se pueden obtener mediante el de un callback llamado en el momento del contacto y grabarlos en un fichero de datos. De este modo parece que puede medirse el pequeño desplazamiento del brazo del dispositivo en la realidad, que surge cuando el puntero penetra en la pieza.

Para la aplicación de una fuerza conocida en el dispositivo háptico, se puede optar por la aplicación del peso de objetos con masas taradas previamente.

Sin embargo, al graficar estos desplazamientos frente a los distintos valores de rigidez, se observa que, a pesar de haber un desplazamiento real en el brazo del dispositivo, Quick Haptics sólo devuelve posiciones del puntero ideales en la superficie del objeto. Esto es debido a que Quick Haptics es una librería de alto nivel pensado para usuarios que sólo desean conocer posiciones en el entorno virtual, no las posiciones suministradas por los encoders del dispositivo.

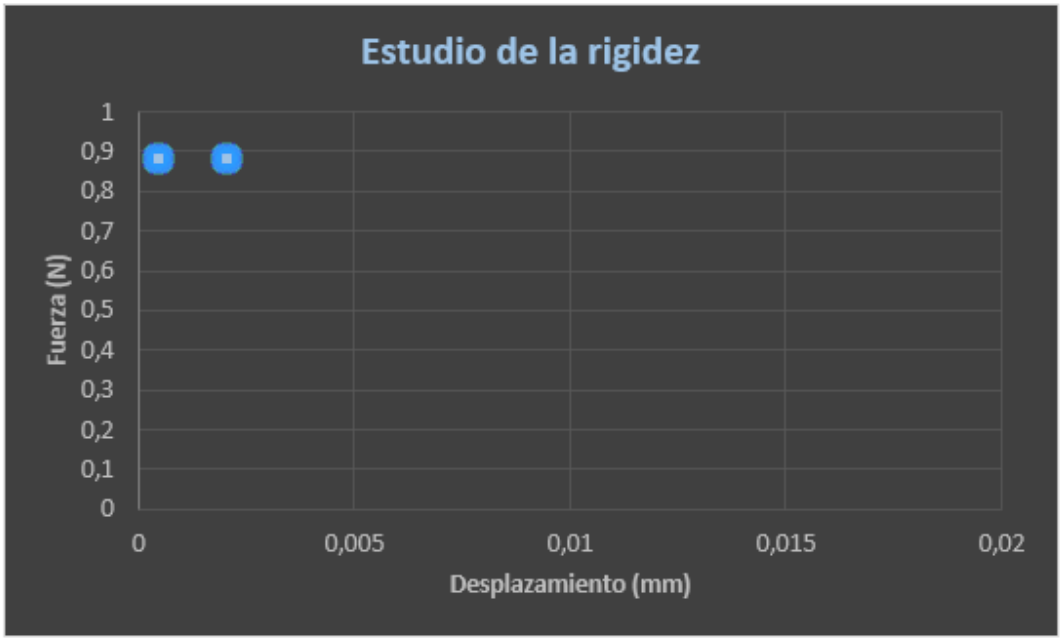


Figura 47. Estudio de la rigidez

Para conseguir unos resultados lo más precisos posibles se leyó la posición y fuerza del dispositivo en cada bucle del hilo principal durante 10 segundos de experimentación, es decir, 10000 valores para cada posición y fuerza. En esta gráfica se han representado todos ellos. Como se puede apreciar la nube de puntos se aglomera en dos zonas principalmente, el motivo de que no sea sólo una es debido al ruido del dispositivo.

Si se observan los órdenes de magnitud del eje de abscisas se llega a la conclusión de que no se corresponde el desplazamiento leído por Open Haptics con el desplazamiento apreciado en la realidad. Reafirmando lo comentado anteriormente y es que Open Haptics no puede devolver la posición exacta del puntero del dispositivo en la realidad.

Debido a este motivo queda claro que es necesario bajar a una librería de más bajo nivel, que sí que pueda proporcionar dicha posición.

En la documentación se mencionan funciones que son capaces de devolver la posición del puntero, pero es necesario bajar hasta una librería de HDAPI, que es la más baja.

Aunque esta sí que puede proporcionar una lectura exacta de la posición del brazo y emplearse junto con la posición del objeto en la escena para aplicar una fuerza proporcional a la distancia entre ellos.

Se puede comenzar así a implementar el cursor háptico manejado por el dispositivo.

3.3 DEFINICIÓN DEL CURSOR HÁPTICO

Previamente se ha explicado el método empleado para el funcionamiento del cursor en el capítulo de ´´modelos hápticos de fuerza de contacto``. Es posible cargar una geometría de una esfera, por ejemplo, que se traslade siguiendo en todo momento la posición dada por cada dispositivo.

También se ha comentado la necesidad de bajar hasta la librería HLAPI para conocer la posición del dispositivo en cada momento. Esto se realiza mediante en un callback y haciendo una llamada a la función HD_CURRENT_POSITION(), que devuelve la posición del dispositivo activo en ese momento. Las funciones que se pueden llamar para obtener datos del dispositivo están mencionadas en el apartado de conexión de múltiples dispositivos. Allí también se explica el motivo de porqué es necesario activar, desactivar y conocer que dispositivo está activado para así obtener datos del mismo.

Posteriormente se hablará del agarre de objetos donde es conveniente el empleo de la función HD_CURRENT_GIMBAL_ANGLES() necesaria para conocer la orientación del lápiz del dispositivo y poder orientar de este modo el objeto agarrado. La función devuelve los ángulos de rotación del dispositivo en radianes.

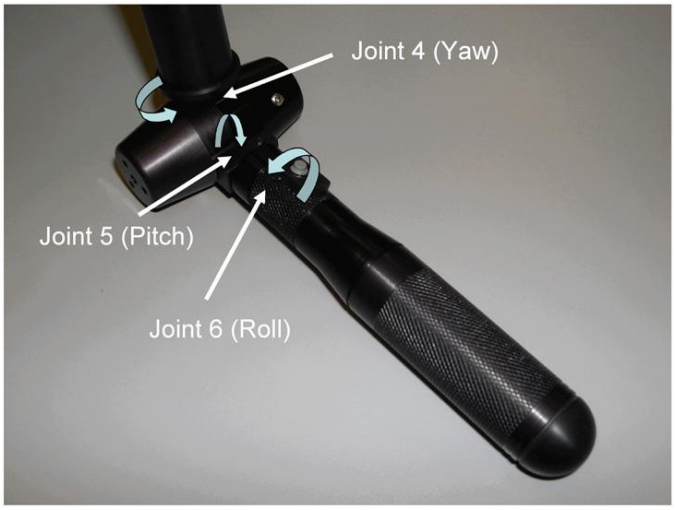


Figura 48. Gimbal angles

Para tener claros los conceptos y la lógica a tener en cuenta en la implementación de un programa HDAPI, como el requerido para el uso del callback que devolverá la respuesta mediante los dispositivos en el momento de un contacto, se deja a continuación un diagrama que lo representa.

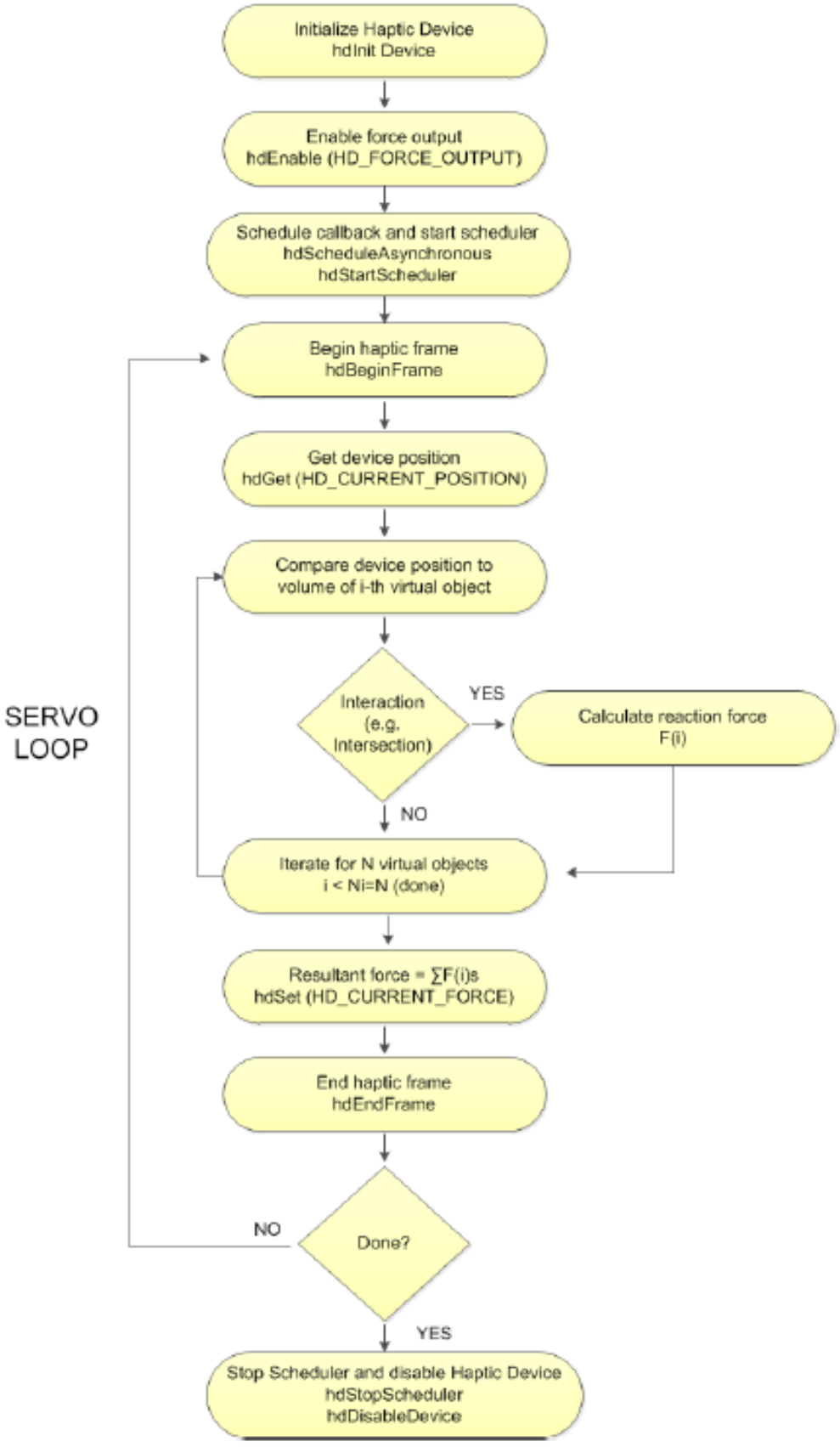

Initialize Haptic Device
hdInit Device
Enable force output
hdEnable (HD_FORCE_OUTPUT)
Schedule callback and start scheduler
hdScheduleAsynchronous
hdStartScheduler
Begin haptic frame
hdBeginFrame
Get device position
hdGet (HD_CURRENT_POSITION)
Compare device position to
volume of i-th virtual object
Interaction
(e.g.
Intersection)
YES
Calculate reaction force
F(i)
NO
Iterate for N virtual objects
i < Ni=N (done)
Resultant force = ∑F(i)s
hdSet (HD_CURRENT_FORCE)
End haptic frame
hdEndFrame
Done?
NO
YES
Stop Scheduler and disable Haptic Device
hdStopScheduler
hdDisableDevice
SERVO
LOOP

Figura 49. Flujo de un programa HDAPI

Para la tarea de seguimiento entre la esfera y el puntero es posible emplear una restricción point to point o ´´punto a punto``, que es básicamente una restricción del tipo contacto esférico, fue explicada en el apartado de restricciones. Se optó por este tipo de restricción debido a que sólo es necesario un único punto que controle la traslación en el espacio de la esfera (no es relevante conocer la orientación de dicha esfera).

Esta restricción se ha puesto entre el centro de masas de la esfera y el puntero del háptico, es decir, la esfera sigue la posición del háptico.

Leer la posición del brazo del háptico en cualquier momento, es posible con funciones ya establecidas que sean llamadas en un callback para ello es conveniente ver el capítulo donde se describe el funcionamiento de múltiples dispositivos hápticos. Aquí se explica que es necesario activar y desactivar continuamente los dispositivos, debido a que sólo es posible controlar uno por cada ciclo del hilo de alta prioridad. En cada llamada que se haga al callback se guardan las posiciones del puntero, hay que tener en cuenta que las coordenadas de estas posiciones no coincidirán con las de Bullet debido a que sus ejes y escalas no son los mismos. Por lo tanto, es necesaria una transformación de las posiciones antes de ser usadas por bullet.

Acto seguido se crea una restricción entre el centro de gravedad de la esfera y esas posiciones leídas y transformadas. Lógicamente existe un desfase entre esos dos puntos, por ejemplo, en el momento de tocar un objeto, el puntero avanzaría, pero la esfera se quedaría en la superficie del objeto. Y es este desfase precisamente el que se usa en la técnica explicada en el apartado de Modelos hápticos de fuerzas de contacto, que básicamente aplica una fuerza proporcional a la distancia esa de desfase y en la dirección normal al contacto.

El método empleado para conocer el punto de contacto será explicado posteriormente.

Cabe mencionar que todo lo explicado anteriormente se ha diseñado pensando en la cooperación de múltiples dispositivos, ya que obviamente sujetar un objeto con la única restricción de un punto es imposible. A continuación, se profundizará más en detalle sobre el agarre de objetos, y ahí se detalla el porqué de la necesidad de conocer no sólo el punto de contacto, si no de la orientación del dispositivo en ese momento.

La problemática de conocer posición, orientación y el punto de contacto no viene dada por obtenerlas, dado que existen funciones predefinidas en Bullet que las proporcionan de forma automática, sino que viene de referenciar de forma correcta respecto al mismo punto de coordenadas todos estos valores.

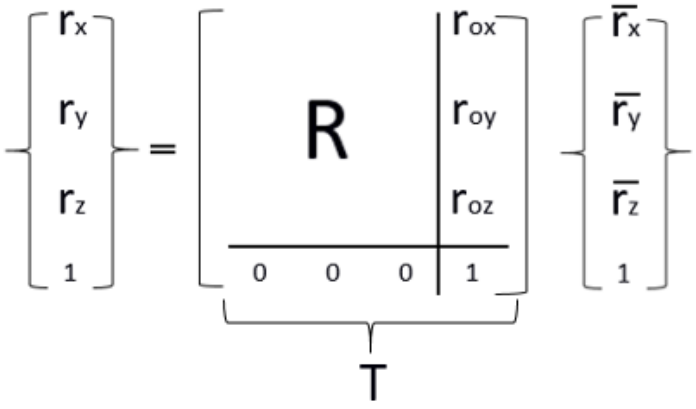
Para el cálculo del punto de agarre es necesario conocer el punto de contacto del puntero con el objeto expresado en coordenadas globales, de tal manera que se pueda calcular la fuerza correspondiente a la posición de ese punto y las posiciones del puntero en instantes posteriores.

Para ello son necesarias una serie de transformaciones y operaciones. Usando una matriz de transformación 4x4 de coordenadas homogéneas, es posible pasar de coordenadas globales a locales y viceversa. Las coordenadas homogéneas consisten en añadir una coordenada homogénea a los vectores de posición de los puntos.

La notación homogénea permite expresar de una manera compacta transformaciones entre distintos sistemas de coordenadas. Una matriz de transformación 4x4, T se compone de un bloque R (tamaño 3x3) que define la rotación entre los sistemas, un vector r0 (tamaño 3x1) que defina la traslación y la coordenada homogénea con valor 1.

La multiplicación matricial de M por un vector cualquiera, calculará la expresión de ese vector transformado.

Entonces, se puede escribir:



Ahora suponiendo un cuerpo cualquiera con centro de masas CDG, visto desde un sistema de referencia global (x,y,z), en una posición aleatoria y con sistema de referencia local (x_0,y_0,z_0) que no tiene porque coincidir con el global. Se pretende conocer la posición de un punto P del objeto en coordenadas globales, conociendo sus coordenadas locales.

Se nombra al vector que apunta a CDG desde el origen: r_{CDG}. El vector que apunta a P desde el origen: r_p. Y por último al vector que apunta a P desde CDG: $\bar{r}_p$.

La representación de todos ellos en el espacio quedaría de la siguiente forma:

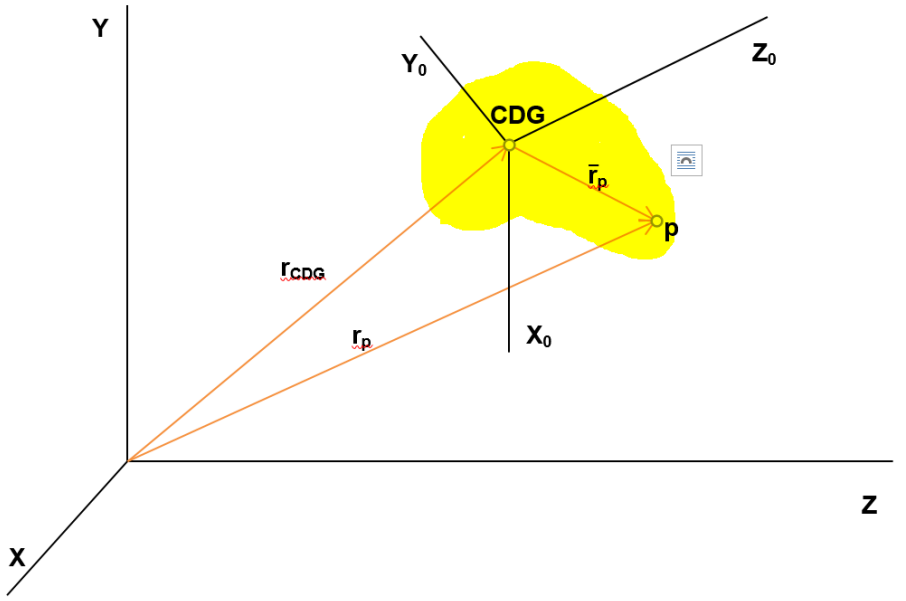


Figura 50. Representación de la situación del objeto con referencias locales y globales

$$r_p = R\,\bar{r}_p + r_{CDG}$$

Esta anterior ecuación se emplea para conocer las coordenadas en el sistema global del punto de agarre, cuyas coordenadas locales son constantes, en cualquier instante.

La posición global se emplea para el cálculo de fuerzas.

$$\bar{r}_p = R^{-1}(r_p - r_{CDG})$$

Esta ecuación, que viene de despejar las coordenadas locales $\bar{r}_p$ en la anterior, realiza la transformación inversa, necesaria dado que las rutinas de detección de contacto proporcionan el punto en el que dos objetos colisionan en el sistema global.

3.4 DETECCIÓN DE CONTACTO

Como ya se explicó en el apartado de información de contacto, es posible obtener información como profundidad de penetración y coordenadas en locales del punto de contacto entre dos objetos que colisionan.

Para ello se hace una llamada al gestor de colisiones btColiisionDispatcher, que, comprobando todos los objetos en la escena, agrupa en manifolds o ´´colectores`` los que colisionan entre sí y devuelve información acerca de ellos.

Sin embargo, existe un problema al intentar aplicarlo a este caso, y es que la información que proporciona el dispatcher no es de que objeto es cual de esos pares de colisión. Es decir, ante una colisión entre la esfera del puntero y un objeto, no se sabría de esos dos objetos que devuelve información cual pertenece a la esfera y cual al objeto. Para solucionar este problema se recurre al uso de IDs.

El uso de IDs consiste en un método similar a las flags y explicadas anteriormente. Se le asigna un número entero a cada objeto, que será su identificador. Posteriormente en el momento que sea necesario, mediante una llamada se puede obtener el identificador del objeto con el que se esté tratando. De este modo es posible conocer que objetos están colisionando y cual es cual.

En este caso se optó por segmentar los IDs en grupos, es decir, para los objetos se empleó identificadores del 1 al 5, y para las esferas 5+i en adelante (donde i es el número de dispositivos).

Para asignar IDs a los objetos, primero se crean y después se les asigna mediante el uso de setUserIndex(), pasándole como argumento el ID que se desee.

```
gateBody[i]->setUserIndex(i);
```

Cuándo se traten las colisiones con el dispatcher se pueden recibir los IDs de cada objeto mediante la llamada:

```
obA->getUserIndex();
```

De este modo ya es posible identificar en un par de objetos que colisionan, a que objeto se hace referencia.

Acto seguido de conocer la detección e identificación de objetos en colisión se estaría en disposición de implementar el agarre de objetos.

3.5 AGARRE DE OBJETOS

Mediante la cooperación de varios dispositivos y la funcionalidad de agarrar objetos con el cursor (restricciones punto a punto con el objeto y la posición obtenida del dispositivo) es posible trasladar y rotar objetos con al menos tres cursores, es decir, tres dispositivos. Aunque para menos dispositivos hápticos, o por comodidad resulta conveniente el uso de restricciones de 6 grados de libertad genéricas, que permiten el agarrar y rotar un objeto al tocarlo con el cursor, para ello es necesario como ya se explicó anteriormente conocer no sólo el punto de contacto si no la posición del dispositivo en ese momento y así poder restringir también los giros de ese agarre.

En la construcción de una nueva restricción de 6 grados de libertad, todos los ejes están bloqueados. Los límites de la restricción btGeneric6DofConstraint se establecen a través de 4 vectores. Dos representan los límites superior e inferior del movimiento angular y los otros dos hacen lo mismo para el movimiento lineal.

- Void setLinearLowerLimit (const btVector3 & linearLower);
- Void setLinearUpperLimit (const btVector3 & linearUpper);

- Void setAngularLowerLimit (const btVector3 & angularLower);
- Void setAngularUpperLimit (const btVector3 & angularUpper);

BtGeneric6DofConstraint tiene los siguientes métodos para establecer límites para cada eje:

- Límite inferior = límite superior (El eje está bloqueado).
- Límite inferior < límite superior (El eje está limitado entre los valores especificados).
- Límite inferior > límite superior (El eje es libre y no tiene límites).

Para la realización de la tarea de agarrar objetos, se llevan a cabo varias operaciones:

1. Se comprueba en un callback si el botón del dispositivo háptico es presionado. Se realiza la comprobación del estado del botón mediante las funciones HD_CUURENT_BUTTONS y HD_LAST BUTTONS.
2. Las colisiones de todos los objetos se analizan y se distinguen cuales corresponden a alguna esfera del puntero, tema tratado en el apartado anterior mediante el uso de IDs.
3. Cuando se cumplen los dos casos anteriores a la vez, se desactivan las restricciones de la esfera y el puntero, se eliminan las representaciones de las esferas y se crean las correspondientes restricciones entre el objeto que tocaba las esferas y el puntero. Para ello es necesario conocer el punto exacto de contacto con el objeto, en posiciones globales y locales respecto a este.
4. Una vez se deja de presionar el botón del dispositivo háptico, vuelve a activarse la geometría de la esfera y restricciones de esta, eliminando previamente las que tuviese el objeto con el puntero.

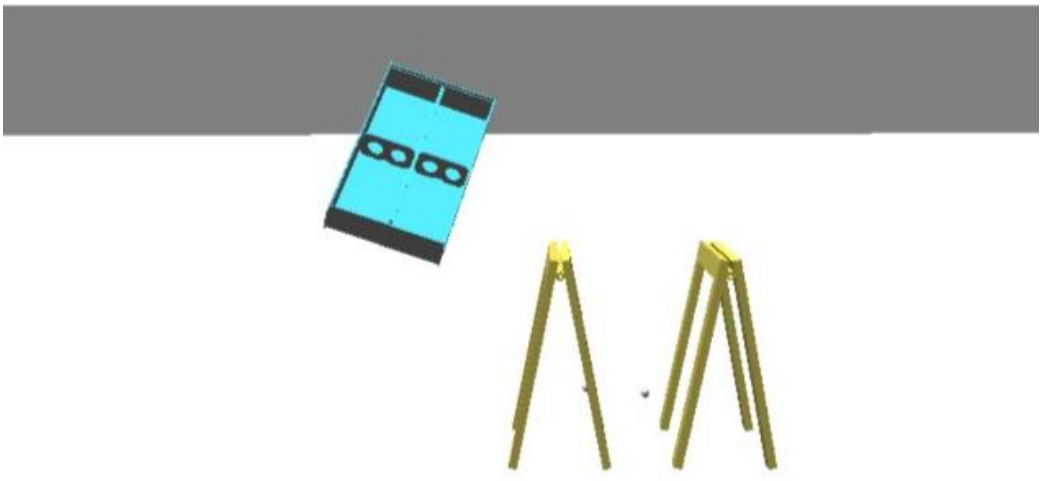


Figura 51. Agarre de chapa metálica

Una vez es posible definir la geometría de los objetos, aplicarles rigidez, definir un cursor háptico, detectar colisiones y agarrar objetos; sólo falta la conexión de múltiples dipositivos que favorezcan la cooperación entre varios usuarios.

3.6 CONEXIÓN MÚLTIPLES DISPOSITIVOS

Cómo ya se ha mencionado anteriormente el último paso para obtener un programa de simulación háptica interactiva, es precisamente la conexión que permita interactuar y coordinar a varios usuarios en la tarea a realizar en la simulación.

Para esta tarea es necesario tener presente todo lo explicado en el apartado conexión de múltiples dispositivos hápticos con Open Haptics.

Para comenzar es conveniente explicar el método a seguir cuando se conecta un dispositivo háptico por primera vez.

En primer lugar, es necesario definir un nombre al dispositivo y asignar el puerto USB al que corresponde. Para ello se puede emplear un toolkit o ´´kit de herramientas`` que trae Open Haptics llamado GEOMAGIC, este permite a los desarrolladores integrar dispositivos hápticos con aplicaciones de terceros existentes y nuevas aplicaciones.

En este kit se incluyen dos programas Geomagic Touch Setup y Geomagic Touch Diagnostic. El primero se emplea para la tarea anteriormente dicha de conectar dispositivos por primera vez. En la siguiente imagen se observa como:

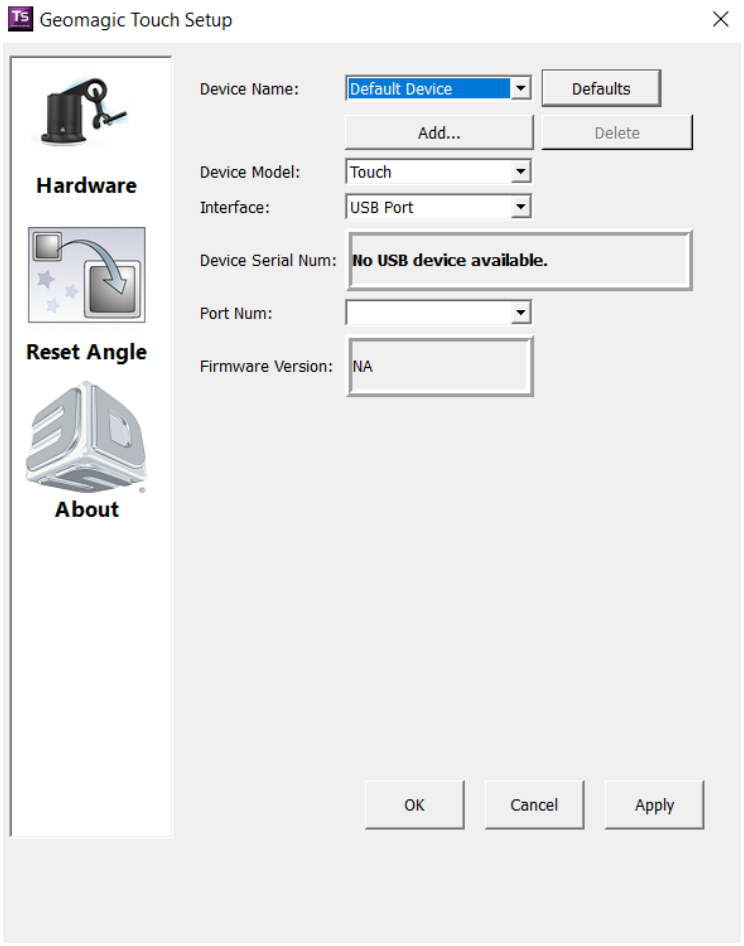


Figura 52. Geomagic Touch Setup

Tras conectar un dispositivo háptico el programa reconoce en que puerto se ha conectado. Lo único que es necesario hacer, es asignar un nombre al dispositivo haciendo click en la casilla Add, en este caso se llamará PHANTOM_1, PHANTOM_2, PHANTOM_3… según el número de dispositivos. Una vez hecho esto se selecciona el puerto al que se refiere cada dispositivo en la casilla Port Num. Por último, se pulsa aplicar y aceptar.

El siguiente paso consiste en calibrar los dispositivos y comprobar la correcta conexión anteriormente realizada. Para ellos se emplea el programa Geomagic Touch Diagnostic.

En este programa clickando en la pestaña select podemos escoger el dispositivo al que se desea hacer referencia según el nombre que se haya escogido en el Geomagic Touch Setup. Una vez hecho esto tenemos acceso a otras pestañas como, por ejemplo, la de calibración. Se muestra a continuación:

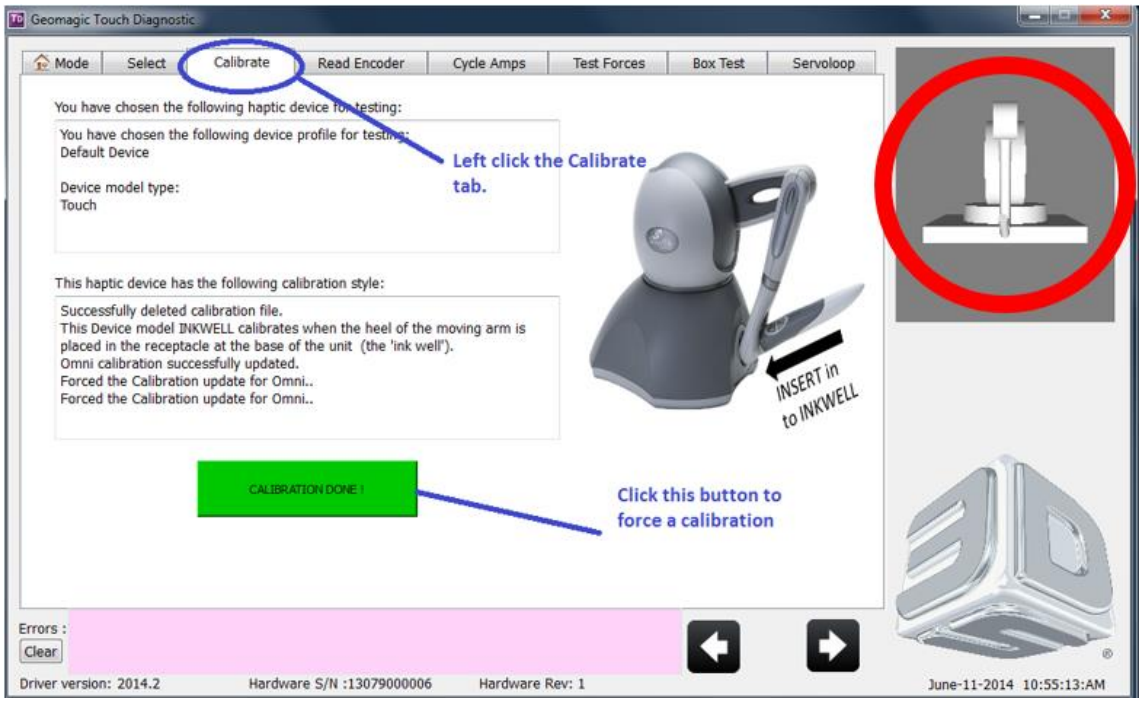


Figura 53. Geomagic Touch Diagnostic calibración

Para el caso de los dispositivos empleados en este estudio la calibración simplemente se efectúa colocando el lápiz en el tintero como aparece descrito en la imagen. Si la calibración se realiza de forma correcta el botón CALIBRATION DONE! se volverá verde como es el caso. También es posible observar la posición del dispositivo en ese momento observando la ventana superior derecha, que la muestra en tiempo real.

Vale la pena señalar que existen otras pestañas como la del servoloop que permite testear la frecuencia a la que está operando dicho servo.

Una vez conectados y calibrados por primera vez los dispositivos se crean los correspondientes contextos del modo que se explica en el apartado conexión de múltiples dispositivos hápticos con QuickHaptics. En él también se hace referencia a la necesidad de tener conocimiento de que dispositivos está activo en cada momento, debido a que sólo es posible el manejo de uno por cada ciclo del loop.

En cada ciclo se comprobará la posición de cada uno (mencionar que es posible conocer más parámetro que la posición) y se asignará una restricción del modo ya explicado anteriormente a cada puntero del respectivo dispositivo con el objeto que se desea mover.

Figura 54. Conexión de múltiples dispositivos

Como se puede observar en la anterior imagen las esferas son movidas cada una por el respectivo dispositivo que tenga emparejado.

4. RESULTADOS

Con el fin de estudiar el comportamiento del programa diseñado, analizando sus respuestas según el tipo de entradas, se han realizado una serie de experimentos. En ellos se han extraído y grabado a fichero valores de la fuerza, desplazamiento y delta (distancia entre el HIP y el PROXY, la definición de estos y como calcular con ellos la fuerza ejercida viene reflejado en el apartado modelos hápticos de fuerza de contacto).

Se han tomado valores una vez por cada vuelta del bucle del hilo principal, es decir, 1000 por segundo. Se han recogido en torno a tiempo t comprendido entre 0 y 10 segundos.

Se puede comprobar que la frecuencia de testeo y número de datos son suficientes para una representación fidedigna de lo ocurrido durante el experimento.

Los experimentos llevados a cabo han sido 4:

1. Colisionar un objeto contra otro a una velocidad constante, analizando la respuesta en fuerza ante ese impacto.
2. Balancear un objeto agarrado para sentir las fuerzas de inercia ejercidas, es decir, que el usuario sienta que está cargando y desplazando un objeto con cierto peso.
3. Intento de penetración de un objeto contra otro hasta llegar a la saturación del dispositivo.
4. Arrastrar el puntero contra una superficie y analizar si su respuesta se corresponde con una fuerza debida al rozamiento a lo largo de esta.

4.1 Colisión de un objeto contra otro a velocidad constante

Este experimento ha consistido en el análisis de la respuesta percibida por el usuario, cuando el dispositivo ejerce unas fuerzas que se oponen al impacto de un objeto contra otro a una velocidad constante.

Para ello se ha decidido agarrar la pieza de chapa metálica e impactarla contra el suelo a una velocidad de caída lo más constante posible. De este modo se puede demostrar que el dispositivo si ejerce una fuerza ante las colisiones de los objetos.

A continuación, se muestra una instantánea de la situación descrita:

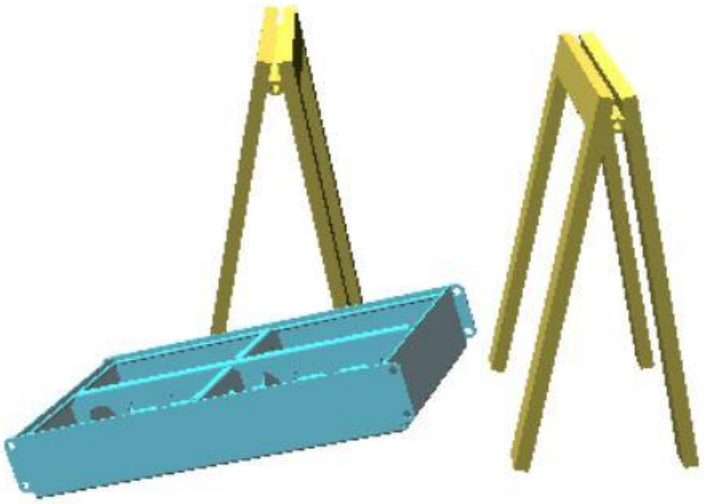


Figura 55. Impacto de un objeto a velocidad constante

A partir de esta situación se extraen y los datos de fuerza, posición del HIP y PROXY, desplazamiento y se procesan calculando la distancia entre el HIP y PROXY. A esa distancia se le llamará delta.

Debido a que en esta operación sólo resulta relevante la componente vertical z de todas las variables leídas. Se ha operado exclusivamente con los datos relacionados con dicha componente.

Si se grafican estos datos frente al tiempo y se comparan entre ellos se obtiene lo siguiente:

Figura 56. Fuerza, delta y desplazamiento en z frente al tiempo para el caso de una colisión a velocidad constante

Analizando la gráfica se llega a una serie de conclusiones:

- El tramo comprendido entre los 0 y 5.5 segundos, refleja la situación en la que el puntero sube hasta la posición de la chapa. Existe una pequeña fuerza no deseada en ese tramo, lo ideal sería que como no existe colisión ni se está arrastrando ningún objeto la fuerza fuese lo menor posible. Sin embargo, se puede afirmar que no resulta ser una magnitud de fuerza que haga del movimiento libre una operación dificultosa ni demasiado perceptible.

- Entre el instante t=5.5s y t=8s la chapa es agarrada y levantada una cierta distancia respecto al suelo en la coordenada z. La fuerza de inercia que aparece al intentar realizar la elevación ya es apreciable, sin embargo, esta fuerza es objeto de estudio del siguiente apartado.

- El objeto cae e impacta contra el suelo entre el tramo t=8s y t=10s. Aquí se observa como durante la caída sigue habiendo una cierta fuerza que se opone al movimiento, también que en cuanto la posición del puntero llega por debajo de cota cero existe un delta que repercute en una fuerza de oposición a dicho desplazamiento. Sin embargo, el pico de fuerza máxima se retrasa un instante como se puede apreciar. Esto puede ser debido a la cantidad de parámetros que tiene que leer el programa y salvar a fichero o incluso debido a una inestabilidad causada por alcanzar la saturación del dispositivo. Este tema será tratado posteriormente en el apartado estudio de intento de penetración.

- A partir de t=10s se producen ciertas oscilaciones en la fuerza debidas el rebote del objeto contra el suelo.

4.2 Estudio de las fuerzas de inercia

Este estudio ha consistido en el análisis de la respuesta percibida por el usuario, cuando el dispositivo ejerce unas fuerzas que se oponen al movimiento cuando se trata de desplazar un objeto agarrado con cierta masa M.

Para ello se ha decidido agarrar la pieza de chapa metálica y balacearla horizontalmente de manera periódica, todo ello a una cierta distancia del suelo para evitar la colisión con algún objeto y tratando de realizar dicho desplazamiento exclusivamente en la componente horizontal. De este modo se puede demostrar que el dispositivo si ejerce una fuerza que hace sentir al usuario que está arrastrando un objeto de cierto peso.

A continuación, se muestra una instantánea de la situación descrita:

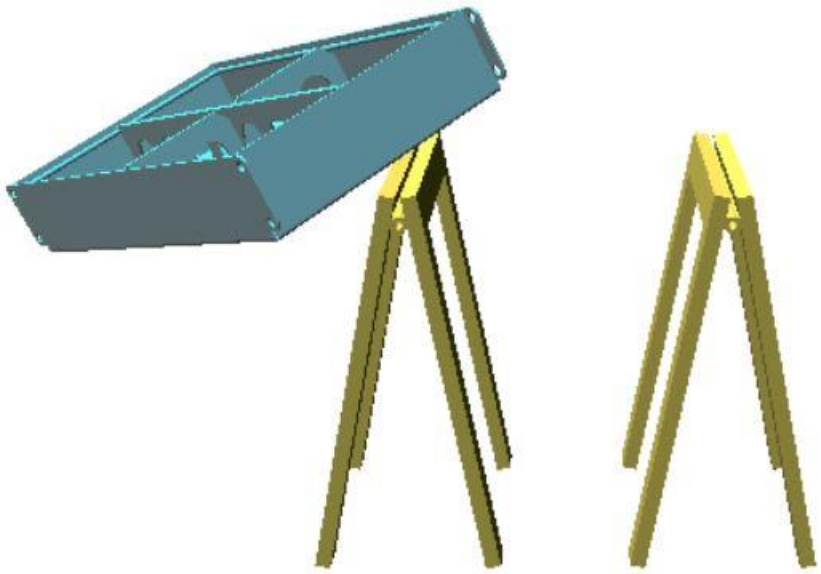


Figura 57. Instantánea del balanceo horizontal que demuestra las fuerzas de inercia ejercidas

A partir de esta situación se extraen y los datos de fuerza, posición del HIP y PROXY, desplazamiento y se procesan calculando la distancia entre el HIP y PROXY. A esa distancia se le llamará delta.

Debido a que en esta operación sólo resulta relevante la componente horizontal x de todas las variables leídas. Se ha operado exclusivamente con los datos relacionados con dicha componente.

Si se grafican estos datos frente al tiempo y se comparan entre ellos se obtiene lo siguiente:

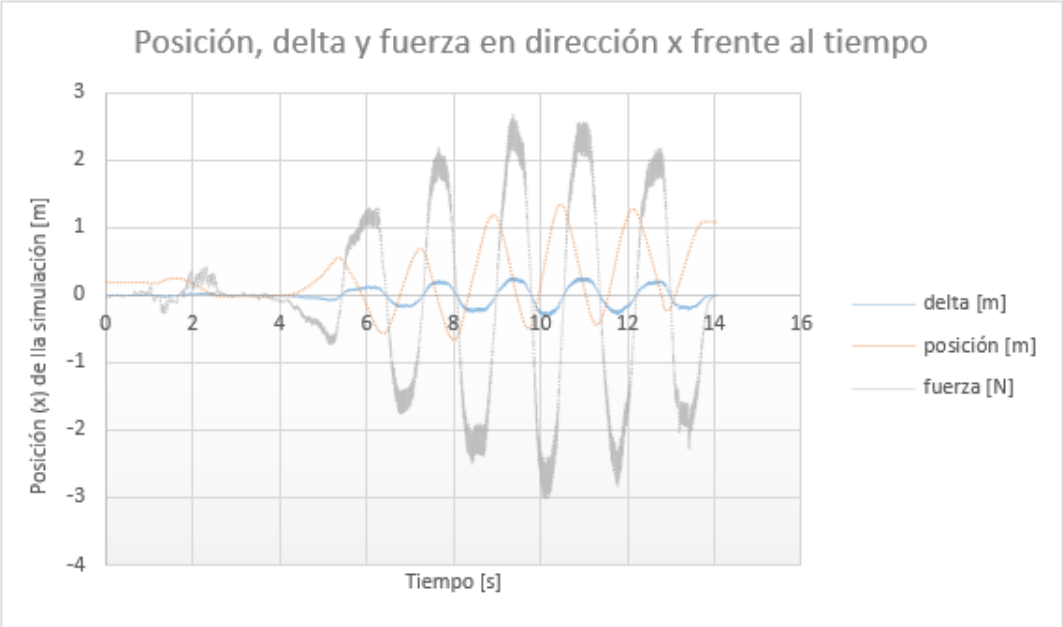


Figura 58. Posición, delta y fuerza en dirección x frente al tiempo para la demostración de la fuerza de inercia

Analizando la gráfica se llega a una serie de conclusiones:

- El tramo comprendido entre los 0 y 4 segundos, refleja la situación en la que el puntero sube hasta la posición de la chapa. Existe una pequeña fuerza no deseada en ese tramo, lo ideal sería que como no existe colisión ni se está arrastrando ningún objeto la fuerza fuese lo menor posible. Sin embargo, se puede afirmar que no resulta ser una magnitud de fuerza que haga del movimiento libre una operación dificultosa ni demasiado perceptible.

- Entre el instante t=4s y t=14s la chapa es agarrada y comienza a balancearse de forma horizontal de un lado a otro de forma periódica a una frecuencia aproximada de 0.5 Hz. Este balanceo causa un delta entre el HIP y PROXY del punto de agarre del objeto que se traduce en una fuerza senoidal aplicada en la componente x del dispositivo. Se puede apreciar, que se trata de un sistema retroalimentado, es decir, el delta induce una fuerza que se opone a dicho desplazamiento y esta fuerza de oposición colabora a que el delta aumente cuando el desplazamiento es en el otro sentido. Esta retroalimentación conlleva a que la fuerza aumente cada vez más. Sin embargo, el usuario controla este fenómeno e induce a su vez una fuerza, ejercida sobre el dispositivo háptico, que impide que la fuerza retroalimentada aumente indefinidamente. Esto se puede observar en el tramo comprendido entre t=10s y t=14s.

4.3 Penetración de un objeto contra otro

Durante el diseño del simulador, fue necesario el conocimiento de que fuerza se podía aplicar en la situación de penetración del puntero o un objeto agarrado contra otro objeto.

Se pudo comprobar que, ensayando con diferentes valores para la constante que hace proporcional la fuerce ejercida cuando existe una diferencia entre el HIP y el PROXY, el valor que mejores resultados obtuvo fue k=10.

Sin embargo, fue necesario conocer la fuerza máxima que el dispositivo era capaz de ejercer sin llegar a dañarse. En la documentación se detalla cómo obtener dichos valores máximos de fuerza. Para ello se emplea la llamada a función nominalMaxForce() en el callback donde se aplican las fuerzas.

El método que se empleó para calcular dicha fuerza fue aplicar directamente la fuerza igual a la constante por la distancia entre HIP y PROXY, pero en el caso de que ese valor de fuerza sea mayor que la fuerza máxima que devuelve la función se aplicará la fuerza máxima exclusivamente. De este modo se limita a un valor máximo y no ejerce indefinidamente una fuerza que se oponga a la penetración.

Para ensayar esta situación se ha decidido intentar penetrar la pieza de chapa metálica con la esfera que mueve el puntero. De este modo se puede demostrar que el dispositivo si ejerce una fuerza acotada ante un penetramiento entre objetos.

A continuación, se muestra una instantánea de la situación descrita:

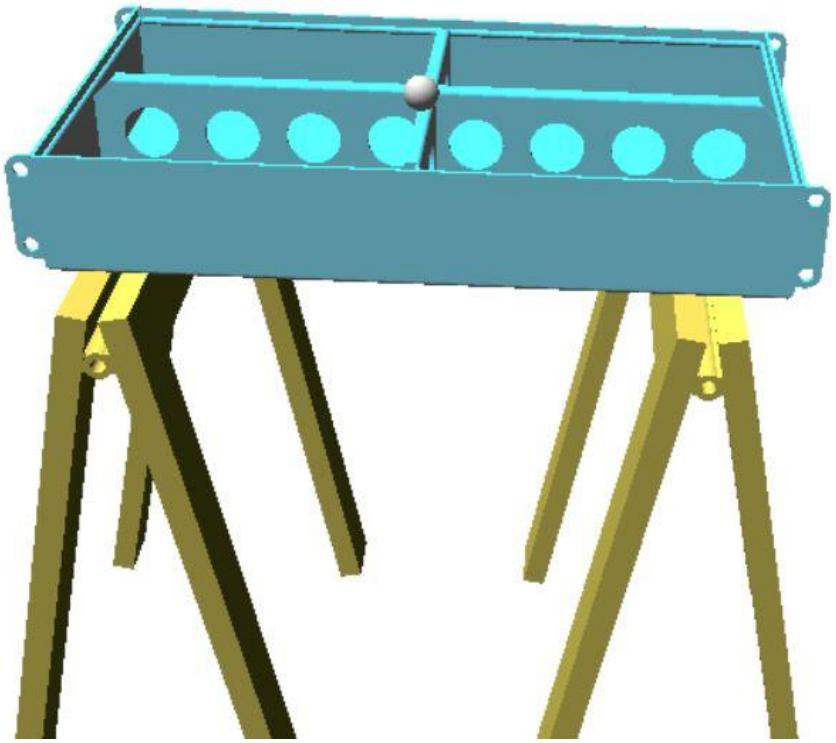


Figura 59. Penetración del puntero contra la chapa metálica

A partir de esta situación se extraen y los datos de fuerza, posición del HIP y PROXY, desplazamiento y se procesan calculando la distancia entre el HIP y PROXY. A esa distancia se le llamará delta.

Como en esta operación resultan relevantes todas las componentes de fuerza y posición, se ha operado haciendo el módulo de las tres componentes de la fuerza y la delta.

Si se grafican estos dos datos se comparan entre ellos, además de cómo varía la delta frente al tiempo, se obtiene lo siguiente:

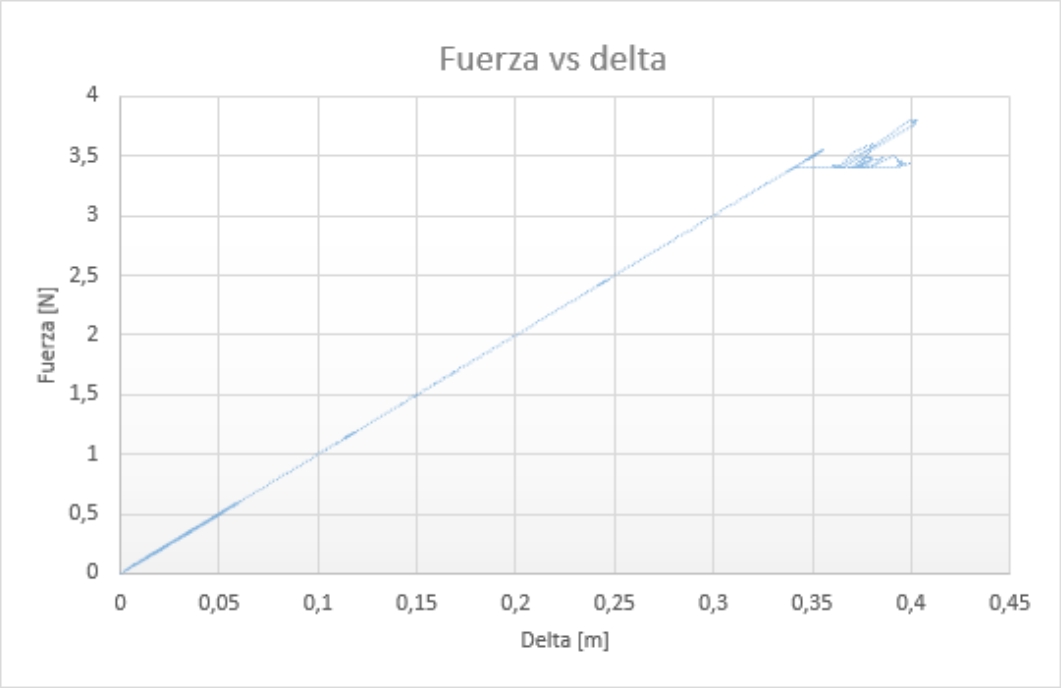


Figura 60. Fuerza vs delta para el caso de intento de penetración de un objeto

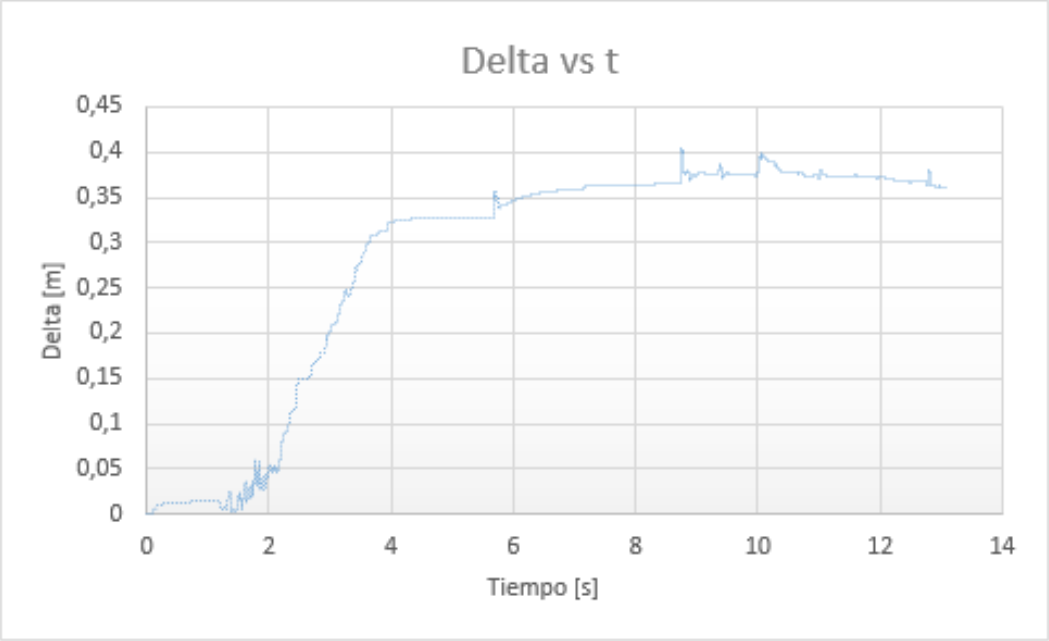


Figura 61. Delta frente al tiempo para el caso de intento de penetración de un objeto

Analizando las gráficas se llega a una serie de conclusiones:

- El tramo comprendido entre los 0 y 2 segundos, refleja la situación en la que el puntero sube hasta la posición de la chapa. Entonces se apoya la esfera sobre la superficie, como la situación no es ideal este instante en el que empieza el contacto causa deltas irregulares.

- Entre el instante t=2s y t=4s se penetra a un ritmo constante en la chapa.

- Al llegar a t=4s la fuerza impide que siga habiendo un incremento del delta.

- A partir de t=4s las inestabilidades producen pequeñas variaciones en la delta, aunque se ve que sigue una tendencia a estar limitada a un valor entorno a 35mm.

- Atendiendo a la gráfica que compara la fuerza con la delta se puede observar el aumento proporcional de la fuerza, el valor de la pendiente de esta recta es la k que se ha mencionado anteriormente. También se aprecia una zona a partir de la cual un aumento de delta no implica un aumento de fuerza. Esta zona refleja que se ha llegado a la zona de saturación dónde el límite de la fuerza es el valor de la fuerza máxima que se ha comentado al principio. Se aprecian ciertas inestabilidades en la fuerza, que pueden ser debidas posiblemente a la lectura y escritura de la cantidad de parámetros de la simulación, o por parte de la fuerza ejercida por el usuario. Sin embargo, se ve que sigue una tendencia a estar limitada entorno a un valor máximo.

Un estudio relacionado con el de penetración, es el del análisis de la respuesta ante el rozamiento que se produce cuando dos cuerpos se friccionan cuando poseen una cierta rugosidad.

A continuación, se detallará más en profundidad como se realizó.

4.4 Rozamiento contra un cuerpo rugoso

Como se ha mencionado este estudio guarda similitud con el visto anteriormente, puesto que también consiste en pequeñas penetraciones que aparecen durante la fricción de dos cuerpos con cierta rugosidad.

La rugosidad es la medida de las variaciones micrométricas en la superficie de los objetos, las cuales les confieren aspereza. Una superficie real por perfecta que aparezca presenta irregularidades que se originan durante el proceso de fabricación. Es por ello que es interesante poder sentirla en la simulación háptica.

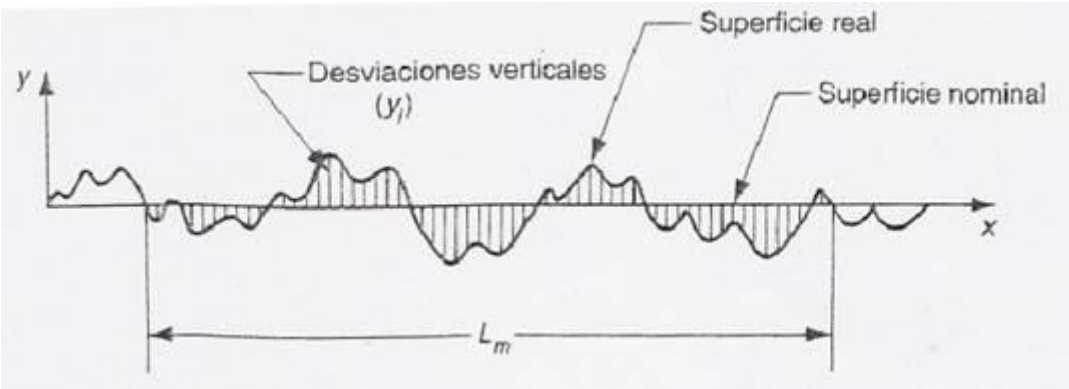


Figura 62. Gráfica rugosidad

El objetivo es obtener una gráfica de datos similar a la anterior en cuanto a desplazamiento del puntero contra la superficie. Y como consecuencia una serie de pequeñas fuerzas que aparezcan por cada salto que se dé entre una arista y otra de las pequeñas deformidades de la superficie. Se obtendrá así una lectura similar a la que realizaría un rugosímetro.

Para la implementación de esto en primer lugar, es necesario conocer el método para la aplicación de cierta rugosidad en la superficie de un objeto. Para ello Bullet ofrece funciones predefinidas que facilitan la tarea. El valor a aplicar de la fricción se les pasa a los cuerpos de los objetos que se desee aplicar una vez hayan sido creados. Se pasa un número comprendido entre 0 y 1, que evalúa la fricción aplicada entre cada par de objetos en una colisión, de todo esto se encarga el dispatcher que fue explicado apartados atrás en la detección de colisiones.

La situación que se ha decidido llevar a cabo consiste en arrastrar la esfera del puntero sobre la superficie del suelo, de tal manera que sea apreciable su rozamiento. Toda esta operación se llevó a cabo tratando de incidir una fuerza constante en la componente

vertical que no falsee los resultados. De este modo se puede demostrar que el dispositivo si ejerce una fuerza que dependerá de la rugosidad del objeto a palpar.

A continuación, se muestra una instantánea de la situación descrita:

Figura 63. Rozamiento de la esfera del puntero contra la superficie del suelo

A partir de esta situación se extraen y los datos de fuerza, posición del HIP y PROXY, desplazamiento y se procesan calculando la distancia entre el HIP y PROXY. A esa distancia se le llamará delta.

Debido a que en esta operación sólo resulta relevante la componente vertical z de todas las variables leídas. Se ha operado exclusivamente con los datos relacionados con dicha componente.

Si se grafican estos datos frente al tiempo y se comparan entre ellos se obtiene lo siguiente:

Figura 64. Fuerza, delta y posición en z frente al tiempo para el caso de rozamiento

Analizando la gráfica se llega a una serie de conclusiones:

- El tramo comprendido entre los 0 y 1 segundo, refleja la situación en la que el puntero baja hasta la cota cero vertical y comienza el contacto contra la superficie. Comienza así a incrementarse el delta a medida que se profundiza por debajo de la superficie. Nótese que la fuerza en la gráfica está dividida entre dos para así poder comparar mejor los tres parámetros, ya que, si no, estarían muy separados.
- Entre t=1s y t=7s se aprecian las pequeñas oscilaciones debidas a la rugosidad del material. Sin embargo, se puede observar una gran cresta debido principalmente a la imposibilidad del usuario de aplicar una fuerza constante en la coordenada vertical mientras arrastra el puntero por la superficie.
- A partir de t=7s la esfera comienza a separarse de la superficie y de este modo se mitiga la fuerza.

5. CONCLUSIONES

Como conclusiones de este estudio se puede concluir que, se ha conseguido diseñar un simulador capaz de: contemplar la casuística relacionada con la simulación de colisiones entre objetos y calcular la dinámica de los cuerpos sólidos en base a ella, siendo capaz de cargar cualquier tipo de geometría diseñada en un programa CAD y simulándola.

También se ha logrado la implementación de dispositivos hápticos en esta simulación, pudiéndose demostrar su comportamiento en el apartado de resultados. Donde se afirma que el usuario es capaz de agarrar objetos, que devuelven fuerzas de oposición al dispositivo háptico en el caso de intentar ser penetrados y también fuerzas de inercia para el caso de ser trasladados por el entorno virtual.

Además, se demostró que el usuario puede palpar la superficie de un objeto y sentir su rugosidad.

El programa realizado permite la conexión de múltiples dispositivos hápticos en un mismo ordenador que permitan la cooperación entre más de un usuario para la realización de una tarea de ensamblaje industrial.

Sin embargo, aún sigue teniendo algunas limitaciones como, por ejemplo, es recomendable el uso de ConvexHull para la carga de geometría, de tal forma que el volumen de colisión quede más ceñido a la geometría del sólido. Existe código ajeno a bullet que es capaz de hacer la descomposición. En este momento la pieza de chapa metálica se comporta como una caja sin ningún tipo de hueco ni orificio. Hubiera sido deseable que simulara la geometría de objetos cóncavos, pero en este proyecto no se ha profundizado hasta tal punto.

En lo referente a emplear una restricción punto a punto entre la esfera y el puntero, se concluye que cuando varios dispositivos sujetan el mismo objeto, se producen inestabilidades que impiden una correcta manipulación de la pieza debido a inestabilidades.

Dichas inestabilidades pueden ser del gran número de parámetros para ajustar el comportamiento correcto de la simulación.

Se ha comprobado que al menos uno de los punteros debe controlar la orientación del objeto mediante una restricción genérica o de 6 grados de libertad. Esto facilita enormemente la tarea de manipular las piezas.

Otro punto a mejorar sería el llevar a cabo restricciones automáticas importadas desde un ensamblaje hecho en CAD, y no ser necesaria la programación manual de estas.

El espacio de trabajo no se ajusta automáticamente dependiendo del punto de vista. Esto puede ser una mejora muy interesante puesto que para realizar operaciones que sean necesarias ser enfocadas desde un punto de vista distinto al de inicialización del entorno es necesario arrastrar con el ratón. Y no varía con la orientación de la cámara el movimiento de los punteros, sigue siendo el mismo que el inicial.

En los resultados se ha podido comprobar la necesidad de una mayor precisión de la respuesta háptica, existen ciertas inestabilidades y oscilaciones leves en el momento de penetración entre el puntero y un objeto.

Por último, también sería una mejora interesante la posibilidad d conexión remota mediante una IP fija, por ejemplo, de varios dispositivos hápticos y así evitar la

necesidad de realizar la simulación y conexión de dispositivos desde el mismo ordenador.

Aún con todo podemos afirmar que la viabilidad de desarrollar un software que alcance los objetivos deseados está demostrada. Con este estudio hemos concluido que un posterior desarrollo de este proyecto llevaría al alcance de un complemento muy ventajoso de un sistema CAD.

6. BIBLIOGRAFÍA

http://www.gmrv.es/~mgarcia/RVA/Hapticos.pdf

http://www.bulletphysics.org/mediawiki-1.5.8/index.php/Collision_Filtering

http://bulletphysics.org/mediawiki-1.5.8/index.php/Activation_States

https://es.wikipedia.org/wiki/Hilo_de_ejecuci%C3%B3n

http://www.ode.org/ode-latest-userguide.html#sec_3_7_0

https://es.wikipedia.org/wiki/Cuaterniones_y_rotaci%C3%B3n_en_el_espacio

https://es.wikipedia.org/wiki/OpenSceneGraph

http://encuentrosjava.uji.es/SegundosEncuentros/materiales/jniworkshop_es.pdf

https://moodle2015-16.ua.es/moodle/pluginfile.php/12375/mod_resource/content/3/vii-11-bullet.pdf

http://www.g-blender.org/foro/

https://www.panda3d.org/manual/index.php/Bullet_Constraints

http://www.bulletphysics.org/mediawiki-1.5.8/index.php/Collision_Filtering

https://en.wikipedia.org/wiki/Polygon_mesh

http://www.geomagic.com/fr/products-landing-pages/haptic/

http://www.g-blender.org/foro/

http://www.geomagic.com/files/8014/3932/7904/OpenHaptics_ProgGuide.pdf

http://www.arscontrol.org/uploads/federica/OpenHaptics%20Toolkit_API%20Reference%20Manual.pdf

http://www.solidworks.com/sw/products/3d-cad/cad-import-export.htm

https://github.com/bulletphysics/bullet3/blob/master/docs/Bullet_User_Manual.pdf

https://github.com/bulletphysics/bullet3/blob/master/docs/BulletQuickstart.pdf

http://www.realtimecollisiondetection.net/

https://www.fiwiki.org/images/f/f5/Dispositivos_Hapticos_y_de_realimentacion_de_fuerza.pdf

http://asignatura.us.es/fgcitig/contenidos/gctem3ma.htm

http://brl.ee.washington.edu/eprints/189/1/Rep242.pdf

https://www.diva-portal.org/smash/get/diva2:509631/FULLTEXT01.pdf

https://arxiv.org/pdf/1701.08879.pdf

http://www.uta.fi/sis/tie/hui/schedule/HUI2011-4-forcefeedback.pdf

https://es.slideshare.net/clow/rugosidad

http://support1.geomagic.com/Support/5605/5668/en-US/Article/View/2377/Haptic-Device-Drivers/0

http://dl.geomagic.com/binaries/support/downloads/Sensable/GeomagicTouchDD_Install_Guide.pdf

http://dl.geomagic.com/binaries/support/downloads/Sensable/3DS/Geomagic-Touch_Device_Guide.pdf

https://en.wikipedia.org/wiki/Wavefront_.obj_file

http://bulletphysics.org/Bullet/phpBB3/viewtopic.php?t=9709

https://es.wikipedia.org/wiki/STL

http://www.g-blender.org/foro/viewforum.php?f=27

http://www.monografias.com/trabajos70/acabados-superficiales-normas-simbologia/acabados-superficiales-normas-simbologia2.shtml

http://www.puntocad.com/index.php/noticias-blog/104-dispositivos-hapticos

UNIVERSIDADE DA CORUÑA

Escola Politécnica Superior

TRABAJO FIN DE GRADO/MÁSTER
CURSO 2016/17

Sistema de comunicación con dispositivos hápticos TOUCH 3D

Grado/Máster en Ingeniería Mecánica

Documento

CÓDIGO

7. APÉNDICE

7.1 CÓDIGO

```
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
#include <osgGA/TrackballManipulator>
#include <osg/MatrixTransform>
#include <osg/ShapeDrawable>
#include <osg/Geode>
#include <stdlib.h>
#include <math.h>
#include <assert.h>
#include <algorithm>
#include <mutex>

#include "BulletCollision/CollisionDispatch/btUnionFind.h"

#include <osgbDynamics/MotionState.h>
#include <osgbDynamics/CreationRecord.h>
#include <osgbDynamics/RigidBody.h>
#include <osgbCollision/CollisionShapes.h>
#include <osgbCollision/RefBulletObject.h>
#include <osgbDynamics/GroundPlane.h>
#include <osgbCollision/GLDebugDrawer.h>
#include <osgbCollision/Utils.h>
#include <osgbInteraction/DragHandler.h>
#include <osgbInteraction/LaunchHandler.h>
#include <osgbInteraction/SaveRestoreHandler.h>

#include <osgwTools/InsertRemove.h>
#include <osgwTools/FindNamedNode.h>
#include <osgwTools/GeometryOperation.h>
#include <osgwTools/GeometryModifier.h>
#include <osgwTools/Shapes.h>

#include <btBulletDynamicsCommon.h>
```

```
#include <osg/io_utils>
#include <osgShadow/ShadowedScene>
#include <osgShadow/LightSpacePerspectiveShadowMap>

#ifndef _MSC_VER
#define stdcall
#endif

#include <HL/hl.h>
#include <HD/hd.h>
#include <HLU/hlu.h>
#include <HDU/hduVector.h>
#include <HDU/hduError.h>

#include <vector>

#ifndef M_PI
#define M_PI 3.14159265358979f
#endif

void HLCALLBACK touchCallback(HDdouble force[3], HLcache*, void*);
void HLCALLBACK startEffectCB(HLcache *, void *);
void HLCALLBACK stopEffectCB(HLcache *, void *);
btVector3 transformPos(const hduVector3Dd &position);

btVector3 btPivotp = { 0, 0, 0 };
const int devices = 1;

std::vector<hduVector3Dd> position(devices);
std::vector<btVector3> positionzero(devices);
std::vector<btVector3> vectorzero(devices);
std::vector<hduVector3Dd> angles(devices);
std::vector<HDboolean> buttonstate(devices, HD_FALSE);
std::vector<HDboolean> griped(devices, HD_FALSE);
std::vector<btVector3> cmposition(devices);
std::vector<btVector3> posdeseada(devices, btVector3(.0, .0, .0));
```

```
HDdouble nominalMaxForce;
float CFM = .1; //parámetros para priorizar restricciones
float ERP = 1;
bool simready = false;

enum CollisionTypes {
	COL_POINTER = 0x1 << 1,
	COL_DEFAULT = 0x1 << 2,
};
// Define filter masks
unsigned int pointerCollidesWith = COL_DEFAULT;
unsigned int defaultCollidesWith = COL_DEFAULT | COL_POINTER;

btRigidBody* makeBody;
std::vector<btRigidBody*> gateBody(5 + devices);

// Pareja de índices de los objetos en colisión.

// Índices de los punteros hápticos y piezas en contacto.
int flagsArray[devices];
std::mutex mutex;

int obj;

struct DeviceContext{
	HHD hHD;
	HHLRC hHLRC;
};

DeviceContext DC[devices];

// Restricciones puntero → esfera
//std::vector<btPoint2PointConstraint*> p2pArray (devices);

std::vector<btGeneric6DofConstraint*> p2pArray(devices);


// Restricciones puntero → objeto agarrado
```

```
//std::vector<btPoint2PointConstraint*> p2ptArray(devices, nullptr);

std::vector<btGeneric6DofConstraint*> p2ptArray(devices, nullptr);


btDiscreteDynamicsWorld* bulletWorld = nullptr;

void initHD(HDstring pConfigName, HHD &hHD)
{
	HDErrorInfo error;

	hHD = hdInitDevice(pConfigName);
	if (HD_DEVICE_ERROR(error = hdGetError()))
	{
		hduPrintError(stderr, &error, "Failed to initialize haptic
device");
		fprintf(stderr, "Press any key to exit");
		getchar();
		exit(-1);
	}

	printf("Found device model: %s / serial number: %s.\n\n",
		hdGetString(HD_DEVICE_MODEL_TYPE),
hdGetString(HD_DEVICE_SERIAL_NUMBER));
}
struct Dispositivo{
	HHD hHD;
	HHLRC hHLRC;
};

static Dispositivo dispositivos[devices];

void calibrate() // funcion que calibrara el dispositivo
{
	if (hdCheckCalibration() == HD_CALIBRATION_NEEDS_UPDATE)
	{
		printf("Calibrando...\n");

		HDint supportedCalibrationStyles;
```

```
            hdGetIntegerv(HD_CALIBRATION_STYLE,
&supportedCalibrationStyles);


            if (supportedCalibrationStyles & HD_CALIBRATION_ENCODER_RESET ||
                  supportedCalibrationStyles & HD_CALIBRATION_INKWELL)
            {
                  printf("Poniendo los encoders a cero... ponga el puntero
en el tintero y presione ENTER.\n");
                  int i;
                  scanf("%i", &i);


            }
            hdUpdateCalibration(supportedCalibrationStyles);
      }
}


btRigidBody *createRigidBody(btDiscreteDynamicsWorld* bw, osg::Node** node,
const osg::Matrix& m, float mass, float friction) // funcion que crea los
cuerpos rigidos a partir de la geometria del objeto
{


      osgwTools::AbsoluteModelTransform* amt = new
osgwTools::AbsoluteModelTransform;
      amt->setDataVariance(osg::Object::DYNAMIC);
      osgwTools::insertAbove(*node, amt);


      osg::ref_ptr< osgbDynamics::CreationRecord > cr = new
osgbDynamics::CreationRecord;
      cr->_sceneGraph = amt;
      cr->_shapeType = CONVEX_HULL_SHAPE_PROXYTYPE; // tipo de mallado que
se emplea
      cr->setCenterOfMass((*node)->getBound().center());
      cr->_parentTransform = m;
      cr->_mass = mass;
      cr->_friction = friction;
      cr->_restitution = .5f;
      cr->setMargin(0.0);              // Eliminación del espaciado en los
contactos.
      cr->_overall = true;             // Necesario para setMargin()
      btRigidBody* rb = osgbDynamics::createRigidBody(cr.get());
```

```
	if (mass<1 && mass>0){
		bw->addRigidBody(rb, COL_POINTER, pointerCollidesWith);
		rb->setActivationState(DISABLE_DEACTIVATION);
	}
	else {
		bw->addRigidBody(rb, COL_DEFAULT, defaultCollidesWith);
		rb->setActivationState(DISABLE_DEACTIVATION);
	}

	// Save RB in global, as AMT UserData (for DragHandler), and in
SaveRestoreHandler.
	makeBody = rb;
	amt->setUserData(new osgbCollision::RefRigidBody(rb));

	*node = amt;
	return makeBody;
}

btDiscreteDynamicsWorld* initPhysics()
{
	btDefaultCollisionConfiguration * collisionConfiguration = new
btDefaultCollisionConfiguration();
	btCollisionDispatcher * dispatcher = new
btCollisionDispatcher(collisionConfiguration);
	btConstraintSolver * solver = new btSequentialImpulseConstraintSolver;

	btVector3 worldAabbMin(-10000, -10000, -10000);
	btVector3 worldAabbMax(10000, 10000, 10000);
	btBroadphaseInterface * inter = new btAxisSweep3(worldAabbMin,
worldAabbMax, 1000);

	btDiscreteDynamicsWorld * dynamicsWorld = new
btDiscreteDynamicsWorld(dispatcher, inter, solver, collisionConfiguration);

	dynamicsWorld->setGravity(btVector3(0, 0, -9.81));

	return(dynamicsWorld);
}

void initializeDevices(){
```

```
	// Initialize HDAPI first, so that the device instances exist in the
system
	// All device instances need to exist before starting the scheduler,
	// which gets started automatically by the first created context
	for(int i = 0; i < devices; i++){
		std::ostringstream oss;
		oss << "PHANTOM_" << i + 1;

		initHD(oss.str().c_str(), DC[i].hHD);
	}
	// Cuando se crea el primer contexto, se arranca automáticamente
	// el planificador, así que ignoraría el resto de los dispositivos
	// que no se hubiesen inicializado hasta entonces.
	for (int i = 0; i < devices; i++)
		DC[i].hHLRC = hlCreateContext(DC[i].hHD);
}


//void switchSphObjConstraint(const int i, btRigidBody *b, const btVector3
&localpos){
//
//	//p2ptArray[i] = new btPoint2PointConstraint(*b, localpos);
//
//	//bulletWorld->addConstraint(p2ptArray[i], true);//se añade la
restriccion punto a punto entre el objeto y el puntero y eliminamos la de la
esfera del puntero
//
//	//p2pArray[i]->setEnabled(false);
//
//}


void switchSphObjConstraint(const int i, btRigidBody *b, const btVector3
&localpos){
	btTransform bt;
	bt.setRotation(btQuaternion(0, 0, 0));
	bt.setOrigin(localpos);
	p2ptArray[i] = new btGeneric6DofConstraint(*b, bt,true);
	bulletWorld->addConstraint(p2ptArray[i], true);//se añade la
restriccion punto a punto entre el objeto y el puntero y eliminamos la de la
esfera del puntero
	p2pArray[i]->setEnabled(false);
}
```

```
//class clampAction: public osgGA::GUIEventHandler{
//     btRigidBody *b;
//public:
//     clampAction(btRigidBody *_b):b(_b){}
//     bool handle(const osgGA::GUIEventAdapter& ea, osgGA::GUIActionAdapter&
aa){
//           if( ea.getEventType() == osgGA::GUIEventAdapter::PUSH )
//           {
//                 if(ea.getButtonMask() &
osgGA::GUIEventAdapter::LEFT_MOUSE_BUTTON){
//                       switchSphObjConstraint(0, b, btVector3(0, 0, 0));
//                       griped[0] = HD_TRUE;
//                       return true;}
//                 if(ea.getButtonMask() &
osgGA::GUIEventAdapter::RIGHT_MOUSE_BUTTON)
//                       griped[0] = HD_FALSE;
//           }
//     }
//};


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


      for (int i = 0; i < devices; i++) // bucle que activara y desactivara
cada dispositivo para pasarle cada funcion
      {
            hlMakeCurrent(DC[i].hHLRC); //funcion que activa el dispositivo
actual
            calibrate();


            hlDisable(HL_USE_GL_MODELVIEW);


            HLuint effect = hlGenEffects(1);


            hlBeginFrame();


            double pzero[3];
            hdGetDoublev(HD_CURRENT_POSITION, pzero);
```

```
            positionzero[i] = transformPos(hduVector3Dd(pzero[0], pzero[1],
pzero[2]));
            printf("pzero %i: %f %f %f\n", i, pzero[0], pzero[1], pzero[2]);
            hdGetDoublev(HD_NOMINAL_MAX_FORCE, &nominalMaxForce);
            hlCallback(HL_EFFECT_COMPUTE_FORCE,
(HLcallbackProc)touchCallback, 0);
            hlCallback(HL_EFFECT_START, (HLcallbackProc)startEffectCB, 0);
            hlCallback(HL_EFFECT_STOP, (HLcallbackProc)stopEffectCB, 0);
            hlStartEffect(HL_EFFECT_CALLBACK, effect);

            hlEndFrame();
      }

      osg::ArgumentParser arguments(&argc, argv);
      const bool debugDisplay(arguments.find("--debug") > 0);

      bulletWorld = initPhysics();
      osg::Group* root = new osg::Group;

      osg::Group* launchHandlerAttachPoint = new osg::Group;
      root->addChild(launchHandlerAttachPoint);


      osg::ref_ptr< osg::Node > rootModel = new osg::Group;


      root->addChild(rootModel.get());

      // Get Node pointers and parent transforms for the wall and gate.
      // (Node names are taken from the osgWorks osgwnames utility.)

      std::vector<osg::Matrix> gateXform(5+devices);

      std::vector<osg::Node*> gateNode(5+devices);

      gateNode[0] = osgDB::readNodeFile("chapa.obj");
      gateNode[1] = osgDB::readNodeFile("caballete.obj");
      gateNode[2] = osgDB::readNodeFile("caballete.obj");
      gateNode[3] = osgDB::readNodeFile("caballete.obj");
```

```
	gateNode[4] = osgDB::readNodeFile("caballete.obj");
	for (int i = 5; i < (devices + 5); i++)
	{
		gateNode[i] = osgDB::readNodeFile("esfera.obj");
	}

	// Posiciones iniciales de los objetos en escena.
	// Chapa
	gateXform[0] = osg::Matrix::rotate(M_PI / 2, 1, 0,
0)*osg::Matrix::translate(0, -0.6, 0.5);
	// Caballetes (2 piezas cada uno)
	gateXform[1] = osg::Matrix::rotate(M_PI / 2, 1, 0,
0)*osg::Matrix::rotate(M_PI / 2, 0, 0, 1)*osg::Matrix::rotate(-M_PI / 16, 0,
1, 0)*osg::Matrix::translate(-0.2, -0.6, 0.1);
	gateXform[2] = osg::Matrix::rotate(M_PI / 2, 1, 0,
0)*osg::Matrix::rotate(-M_PI / 2, 0, 0, 1)*osg::Matrix::rotate(M_PI / 16, 0,
1, 0)*osg::Matrix::translate(-0.2, -0.6, 0.1);
	gateXform[3] = osg::Matrix::rotate(M_PI / 2, 1, 0,
0)*osg::Matrix::rotate(M_PI / 2, 0, 0, 1)*osg::Matrix::rotate(-M_PI / 16, 0,
1, 0)*osg::Matrix::translate(0.2, -0.6, 0.1);
	gateXform[4] = osg::Matrix::rotate(M_PI / 2, 1, 0,
0)*osg::Matrix::rotate(-M_PI / 2, 0, 0, 1)*osg::Matrix::rotate(M_PI / 16, 0,
1, 0)*osg::Matrix::translate(0.2, -0.6, 0.1);

	osg::ref_ptr< osgbInteraction::SaveRestoreHandler > srh = new
	osgbInteraction::SaveRestoreHandler;

	for (int i = 0; i < (devices + 5); i++)
	{
		rootModel->asGroup()->addChild(gateNode[i]);
		if (gateNode[i] == NULL){
			return(1);
		}
		if ( i < 5)
		{
		gateBody[i] = createRigidBody(bulletWorld, &gateNode[i],
gateXform[i], 1, 1);
		gateBody[i]->setUserIndex(i);
		}
		if (i >= 5 && i < (5 + devices))
		{
```

```
                    gateXform[i] = osg::Matrix::translate((6.5 - i) / 16, 0.5,
0);
                    vectorzero[i - 5][0] = ((6.5 - i) / 16) - positionzero[i -
5][0];
                    vectorzero[i-5][1] = 0 - positionzero[i - 5][1];
                    vectorzero[i-5][2] = 0 - positionzero[i-5][2];
                    printf("pzero %i: %d %d %d\n", i, vectorzero[i - 5][0],
vectorzero[i - 5][1], vectorzero[i - 5][2]);
                    gateBody[i] = createRigidBody(bulletWorld, &gateNode[i],
gateXform[i], 0.005, 1.4);
                    gateBody[i]->setUserIndex(i);
             }
      }

      // Add ground
      const osg::Vec4 plane(0., 0., 1., 0.);
      osg::Node *solidPlane = osgbDynamics::generateGroundPlane(plane,
bulletWorld, NULL, COL_DEFAULT, defaultCollidesWith);
      solidPlane->setNodeMask(2);      // Recibe sombras
      root->addChild(solidPlane);



      // Create the hinge constraint.
      {
             const btVector3 btPivot(0, 0.217431992, -0.0100280046);
             btVector3 btAxisA(1, 0, 0);
             btVector3 btAxisB(-1, 0, 0);
             btHingeConstraint* hinge = new btHingeConstraint(*gateBody[1],
*gateBody[2], btPivot, btPivot, btAxisA, btAxisB);
             btHingeConstraint* hinge1 = new btHingeConstraint(*gateBody[3],
*gateBody[4], btPivot, btPivot, btAxisA, btAxisB);
             bulletWorld->addConstraint(hinge, true);
             bulletWorld->addConstraint(hinge1, true);
             hinge->setLimit(-M_PI / 8, M_PI / 96);
             hinge1->setLimit(-M_PI / 8, M_PI / 96);
      }

      // Create the point to point constraints.
      for(int i = 0; i < devices; i++){
             /*p2pArray[i] = new btPoint2PointConstraint(*gateBody[5 + i],
btPivotp);
```

```
            bulletWorld->addConstraint(p2pArray[i], true);
            p2pArray[i]->setParam(BT_P2P_FLAGS_CFM, CFM);
            p2pArray[i]->setParam(BT_P2P_FLAGS_ERP, ERP);*/
            btTransform t;
            t.setRotation(btQuaternion(0, 0, 0));
            t.setOrigin(btVector3 (0,0,0));
            p2pArray[i] = new btGeneric6DofConstraint(*gateBody[5 + i], t,
true);
            bulletWorld->addConstraint(p2pArray[i], true);
            /*p2pArray[i]->setParam(BT_6DOF_FLAGS_CFM_STOP, CFM);
            p2pArray[i]->setParam(BT_6DOF_FLAGS_ERP_STOP, ERP); */
      }

      osgbCollision::GLDebugDrawer* dbgDraw(NULL);
      /*if (true)*/
      if (debugDisplay)
      {
            dbgDraw = new osgbCollision::GLDebugDrawer();
            dbgDraw->setDebugMode(~btIDebugDraw::DBG_DrawText);
            bulletWorld->setDebugDrawer(dbgDraw);
            root->addChild(dbgDraw->getSceneGraph());
      }

      osgViewer::Viewer viewer(arguments);
      viewer.setUpViewInWindow(30, 30, 768, 480);
      viewer.setSceneData(root);

      osgGA::TrackballManipulator* tb = new osgGA::TrackballManipulator;
      tb->setHomePosition(osg::Vec3(0., -3., 1.), osg::Vec3(0., 0., 0.),
osg::Vec3(0., 0., 0.));
      viewer.setCameraManipulator(tb);
      viewer.getCamera()->setClearColor(osg::Vec4(.5, .5, .5, 1.));
      viewer.realize();

      // Create the launch handler.
      osgbInteraction::LaunchHandler* lh = new
osgbInteraction::LaunchHandler(
      bulletWorld, launchHandlerAttachPoint, viewer.getCamera());
      {
```

```
            // Use a custom launch model: Sphere with radius 0.1 (instead of
default 1.0).
            osg::Geode* geode = new osg::Geode;
            const double radius(.1);
            geode->addDrawable(osgwTools::makeGeodesicSphere(radius));
            lh->setLaunchModel(geode, new btSphereShape(radius));
            lh->setInitialVelocity(20.);

            // Also add the proper collision masks
            lh->setCollisionFilters(COL_DEFAULT, defaultCollidesWith);

            viewer.addEventHandler(lh);
      }

      srh->setLaunchHandler(lh);
      srh->capture();
      viewer.addEventHandler(srh.get());
      viewer.addEventHandler(new osgbInteraction::DragHandler(
      bulletWorld, viewer.getCamera()));

      {
            osgShadow::ShadowedScene *sh= new osgShadow::ShadowedScene;
            osgShadow::LightSpacePerspectiveShadowMapVB *sm = new
osgShadow::LightSpacePerspectiveShadowMapVB;

            sm->setLight(viewer.getLight());
            sm->setTextureSize(osg::Vec2s(1024, 1024));
            osg::LightSource *ls = new osg::LightSource;
            ls->setLight(viewer.getLight());

            sh->setReceivesShadowTraversalMask(2);
            sh->setCastsShadowTraversalMask(1);
            sh->setShadowTechnique(sm);
            sh->addChild(viewer.getSceneData());
            sh->addChild(ls);

            viewer.setSceneData(sh);          //asignamos al viewer la raíz
del modelo MBS (incluye los grupos Ground y Bodies y las sombras)
```

```
		//Luces y datos para render.
		viewer.setLightingMode(osg::View::SKY_LIGHT);
		viewer.getLight()->setAmbient(osg::Vec4(0.2, 0.2, 0.2, 1.0));
		viewer.getLight()->setDiffuse(osg::Vec4(1.0, 1.0, 1.0, 1.0));
		viewer.getLight()->setSpecular(osg::Vec4(1.0, 1.0, 1.0, 1.0));
		viewer.getLight()->setPosition(osg::Vec4(-50.0, -50.0, 150.0,
0.0));
	}

	simready = true;
	double prevSimTime = 0.;
	while (!viewer.done())
	{
		if (dbgDraw != NULL)
			dbgDraw->BeginDraw();

		const double currSimTime = viewer.getFrameStamp()-
>getSimulationTime();

		bulletWorld->stepSimulation(currSimTime - prevSimTime);
		prevSimTime = currSimTime;

		if (dbgDraw != NULL)
		{
			bulletWorld->debugDrawWorld();
			dbgDraw->EndDraw();
		}
		viewer.frame();

		int numManifolds = bulletWorld->getDispatcher()-
>getNumManifolds();

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

			if (griped[i] == HD_FALSE && buttonstate[i] == HD_TRUE){

				for (int j = 0; j < numManifolds; j++)
				{
```

```
                                btPersistentManifold* contactManifold =
bulletWorld->getDispatcher()->getManifoldByIndexInternal(j);
                                // se pueden recibir las flags de los pares
de objetos que colisionan, pero no que flag corresponde a cada uno
                                const btCollisionObject* obA =
contactManifold->getBody0();
                                const btCollisionObject* obB =
contactManifold->getBody1();
                                int numContacts = contactManifold-
>getNumContacts();
                                {
                                      for (int k = 0; k < numContacts; k++)
                                      {
                                            auto indx = std::make_pair(obA-
>getUserIndex(), obB->getUserIndex());
                                            //mutex.lock();

                                            if (indx.first == (5 + i) ||
indx.second == (5 + i)){

                                                  btManifoldPoint& pt =
contactManifold->getContactPoint(k);
                                                  if (pt.getDistance() <
0.f){
                                                        btVector3 pts[2];
                                                        pts[0] =
pt.getPositionWorldOnA();
                                                        pts[1] =
pt.getPositionWorldOnB();
                                                        flagsArray[i] =
indx.first == 5 + i ? indx.second : indx.first;
                                                        btRigidBody *b =
gateBody[flagsArray[i]];
                                                        btVector3 localpos
= b->getCenterOfMassTransform().invXform(pts[obj]);

      switchSphObjConstraint(i, b, localpos);
                                                        printf("Uniendo
cuerpo %d, dispositivo %d\n", flagsArray[i], i);
                                                        griped[i] =
HD_TRUE;
                                                        gateBody[(5+i)]-
>forceActivationState(DISABLE_SIMULATION);
                                                        gateNode[(5+i)]-
>setNodeMask(0);
```

```
                                                goto salida;
                                            }
                                        }
                                        //mutex.unlock();
                                    }
                                }
                            }
                        }
                        salida:
                        if (p2ptArray[i]!=nullptr && !buttonstate[i]){
                            p2ptArray[i]->setEnabled(false);
                            bulletWorld->removeConstraint(p2ptArray[i]);
                            printf("borrando: %p\n", p2ptArray[i]);
                            delete p2ptArray[i];
                            p2ptArray[i] = nullptr;
                            p2pArray[i]->setEnabled(true);
                            gateBody[(5+i)]-
>forceActivationState(DISABLE_DEACTIVATION);//cuando se suelta el objeto
eliminamos la restriccion  del objeto y volvemos a activar la de la esfera
                            gateNode[(5+i)]->setNodeMask(1);
                            griped[i] = HD_FALSE;
                        }
                        if (griped[i] == HD_TRUE){
                            /*p2ptArray[i]->setPivotB(posdeseada[i]);*/
                            btRigidBody *b = gateBody[flagsArray[i]];
                            btVector3 localpos = b-
>getCenterOfMassTransform().invXform(posdeseada[i]);
                            p2ptArray[i]->setLimit(0, localpos[0],
localpos[0]);
                            p2ptArray[i]->setLimit(1, localpos[1],
localpos[1]);
                            p2ptArray[i]->setLimit(2, localpos[2],
localpos[2]);
                            p2ptArray[i]->setLimit(3, angles[i][1],
angles[i][1]);
                            p2ptArray[i]->setLimit(4, angles[i][2],
angles[i][2]);
                            p2ptArray[i]->setLimit(5, angles[i][0],
angles[i][0]);

                        }
```

```
                    /*else p2pArray[i]->setPivotB(posdeseada[i]);*/
                    else {
                          btRigidBody *b = gateBody[flagsArray[i]];
                          btVector3 localpos = b-
>getCenterOfMassTransform().invXform(posdeseada[i]);
                          p2pArray[i]->setLimit(0, localpos[0], localpos[0]);
                          p2pArray[i]->setLimit(1, localpos[1], localpos[1]);
                          p2pArray[i]->setLimit(2, localpos[2], localpos[2]);
                          p2pArray[i]->setLimit(3, 0, 0);
                          p2pArray[i]->setLimit(4, 0, 0);
                          p2pArray[i]->setLimit(5, 0, 0);
                          printf("localpos: %f %f %f \n", localpos[0],
localpos[1], localpos[2]);
                    }
             }
      }

      return(0);
}

void HLCALLBACK touchCallback(HDdouble force[3], HLcache* cache, void*)
//evento que sera llamado cada vez que exista contacto con el dispositivo
haptico
{
      static int counter = 0;

      if (!simready){
            force[0] = 0;
            force[1] = 0;
            force[2] = 0;
            return;
      }
      HDint nCurrentButtons, nLastButtons;
      HHD hHDactual = hdGetCurrentDevice();
      hduVector3Dd CDG;
      hduVector3Dd delta;

      for (int i = 0; i < devices; i++){
            if (hHDactual == DC[i].hHD){
                  hdBeginFrame(hHDactual);
```

```
                    hdGetDoublev(HD_CURRENT_POSITION, &position[i][0]);
                    hdGetDoublev(HD_CURRENT_GIMBAL_ANGLES, &angles[i][0]);
                    hdGetIntegerv(HD_CURRENT_BUTTONS, &nCurrentButtons);
                    hdGetIntegerv(HD_LAST_BUTTONS, &nLastButtons);
                    hdEndFrame(hHDactual);
                    posdeseada[i] =
vectorzero[i]+transformPos(hduVector3Dd(position[i]));

                    if ((nCurrentButtons & HD_DEVICE_BUTTON_1) != 0 &&
                           (nLastButtons & HD_DEVICE_BUTTON_1) == 0)
                    {
                           /* Detected button down */
                           buttonstate[i] = HD_TRUE;
                    }
                    else if ((nCurrentButtons & HD_DEVICE_BUTTON_1) == 0 &&
                           (nLastButtons & HD_DEVICE_BUTTON_1) != 0)
                    {
                           /* Detected button up */
                           buttonstate[i] = HD_FALSE;
                    }
                    if (griped[i] == HD_TRUE){
                           mutex.lock();
                           auto b = gateBody[flagsArray[i]];
                           cmposition[i] = b->getCenterOfMassPosition();
                           /*printf("cmpos: %f %f %f \n", cmposition[i][0],
cmposition[i][1], cmposition[i][2]);*/
                           mutex.unlock();
                           }
                    if (griped[i] == HD_FALSE){
                           cmposition[i] = gateBody[5 + i]-
>getCenterOfMassPosition();
                           /*printf("cmpos: %f %f %f \n", cmposition[i][0],
cmposition[i][1], cmposition[i][2]);*/
                           }
                    CDG = hduVector3Dd(cmposition[i][0], cmposition[i][1],
cmposition[i][2]);
                    delta = CDG - hduVector3Dd(posdeseada[i][0],
posdeseada[i][1], posdeseada[i][2]);
                    //printf("posde: %f %f %f \n", posdeseada[i][0],
posdeseada[i][1], posdeseada[i][2]);
             }
```

```
	} // se ha comprobado si el boton del dispositivo estaba pulsado y realizada la casuistica entre posiciones deseadas para punteros y objetos


	float modulodelta = delta.magnitude();
	const HDdouble stepSize = 0.8;
	const double k = 10;
	hduVector3Dd forceVec(0, 0, 0);


	if (modulodelta > stepSize) // condicion que evitara desestabilidades ante posiciones que obliguen a realizar fuerzas bruscas para llegar a la posicion deseada
	{
		delta.normalize();
		forceVec = k*stepSize * delta;
	}
	else
	{
		forceVec = k*delta;
	}


	for (int i = 0; i<3; i++)
	{
		// Forma estándar de calcular que -nominalMaxForce ≤ forceVec[i] ≤ nominalMaxForce
		forceVec[i] = std::max(std::min(forceVec[i], nominalMaxForce), -nominalMaxForce);
	}


	force[0] = forceVec[0];
	force[2] = -forceVec[1];
	force[1] = forceVec[2];
}
btVector3 transformPos(const hduVector3Dd &position) //funcion que hace las transformaciones necesarias a la orientacion de los ejes para que coincidan y tambien escala las posiciones
{
	btVector3 pd;
	pd[0] = 0.01*position[0];
	pd[1] = -1 - 0.025*position[2];
	pd[2] = 0.04*position[1];
	return(pd);
```

```
}
void HLCALLBACK startEffectCB(HLcache *cache, void *)
{
}
void HLCALLBACK stopEffectCB(HLcache *cache, void *)
{
}
```

UNIVERSIDADE DA CORUÑA

Escola Politécnica Superior

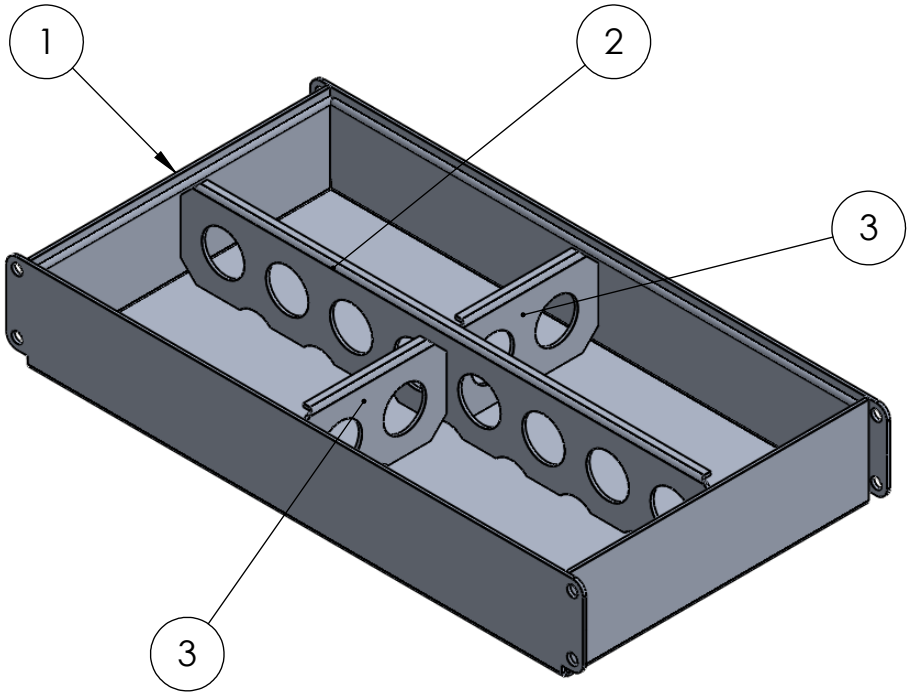
TRABAJO FIN DE GRADO/MÁSTER
CURSO 2016/17

Sistema de comunicación con dispositivos hápticos TOUCH 3D

Grado/Máster en Ingeniería Mecánica

Documento

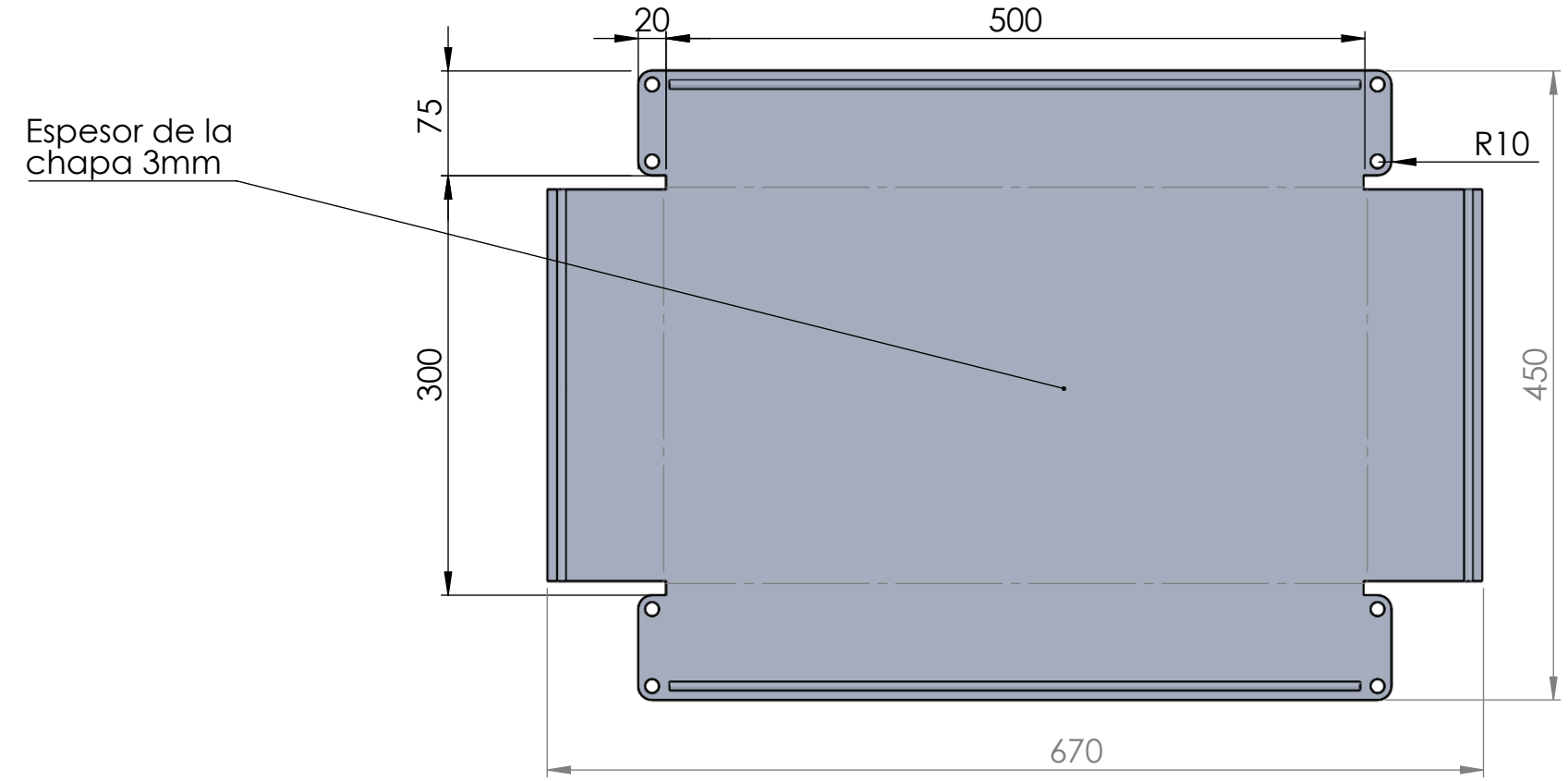
PLANOS

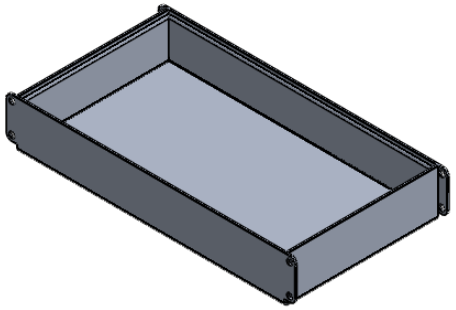


N.º DE ELEMENTO	N.º DE PIEZA	CANTIDAD
1	Pieza chapa metálica exterior	1
2	Pieza chapa metálica interior 2	1
3	Pieza chapa metálica interior 1	2

SOLIDWORKS Student Edition.
Solo para uso académico.

SI NO SE INDICA LO CONTRARIO: LAS COTAS SE EXPRESAN EN MM ACABADO SUPERFICIAL: TOLERANCIAS: LINEAL: ANGULAR:		ACABADO:		REBARBAR Y ROMPER ARISTAS VIVAS	NO CAMBIE LA ESCALA		REVISIÓN
	NOMBRE	FIRMA	FECHA				TÍTULO:
DIBUJ.	Aitor García Catoira		16/07/17				
VERIF.							
APROB.							
FABR.							
CALID.				MATERIAL: Aluminio		N.º DE DIBUJO Pieza chapa metálica	A4
				PESO:		ESCALA:1:5	HOJA 1 DE 6

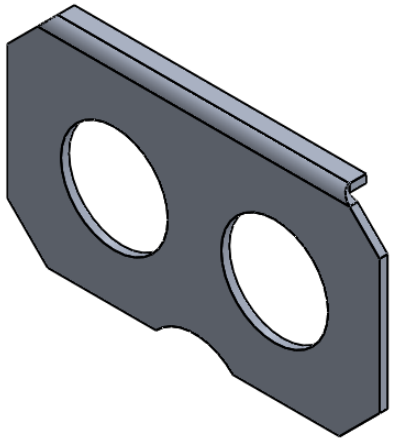


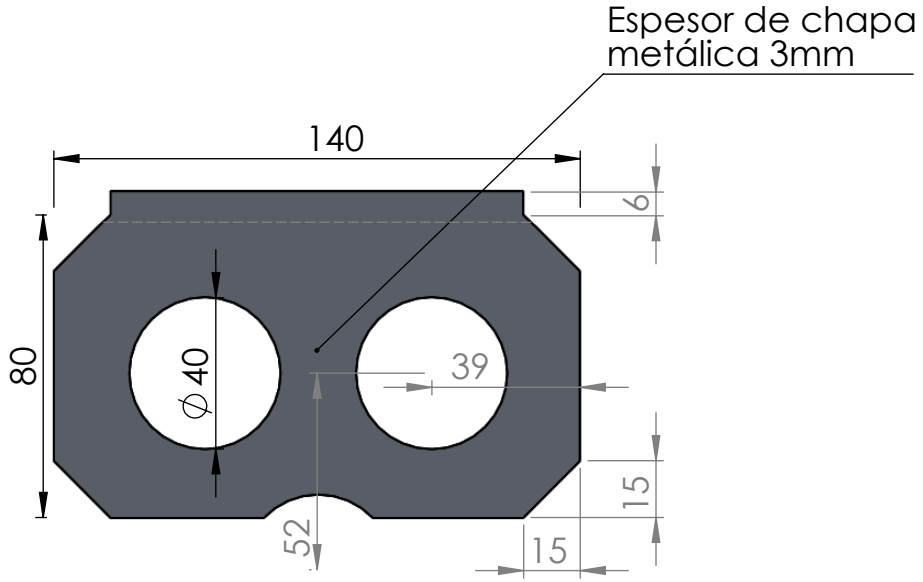


Escala 1:10

SOLIDWORKS Student Edition.
Solo para uso académico.

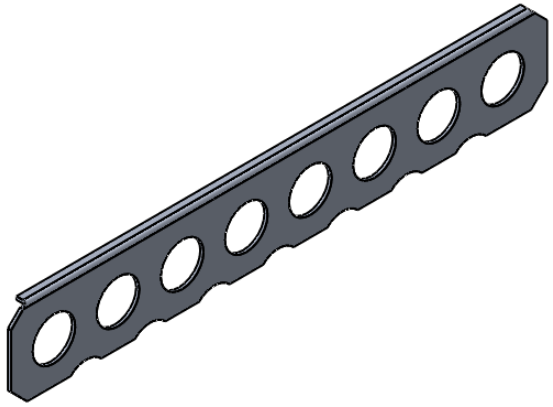
SI NO SE INDICA LO CONTRARIO: LAS COTAS SE EXPRESAN EN MM ACABADO SUPERFICIAL: TOLERANCIAS: LINEAL: ANGULAR:	ACABADO:		REBARBAR Y ROMPER ARISTAS VIVAS		NO CAMBIE LA ESCALA	REVISIÓN
	NOMBRE	FIRMA	FECHA			TÍTULO:
DIBUJ.	Aitor García Catoira		16/07/17			
VERIF.						
APROB.						
FABR.						
CALID.				MATERIAL: Aluminio		N.º DE DIBUJO: Pieza chapa metálica exterior
						A4
				PESO:	ESCALA:1:5	HOJA 2 DE 6

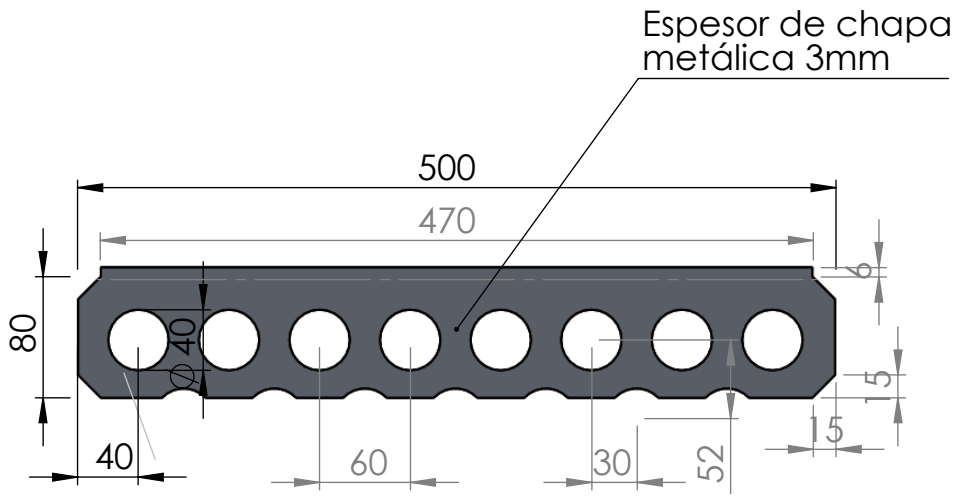




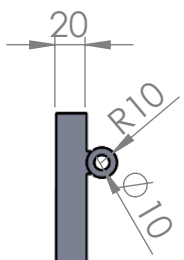
SOLIDWORKS Student Edition.
Solo para uso académico.

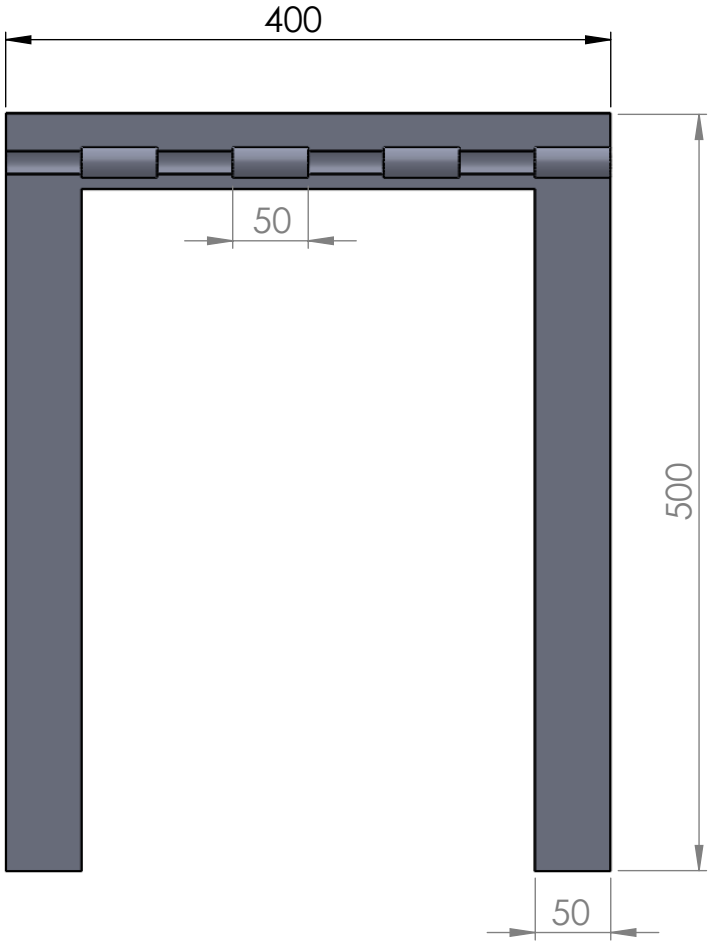
SI NO SE INDICA LO CONTRARIO: LAS COTAS SE EXPRESAN EN MM ACABADO SUPERFICIAL: TOLERANCIAS: LINEAL: ANGULAR:		ACABADO:			REBARBAR Y ROMPER ARISTAS VIVAS	NO CAMBIE LA ESCALA	REVISIÓN
	NOMBRE	FIRMA	FECHA			TÍTULO:	
DIBUJ.	Aitor García Catoira		16/07/17				
VERIF.							
APROB.							
FABR.							
CALID.				MATERIAL: Aluminio		N.º DE DIBUJO Pieza chapa metálica interior 1	A4
				PESO:		ESCALA:1:2	HOJA 3 DE 6

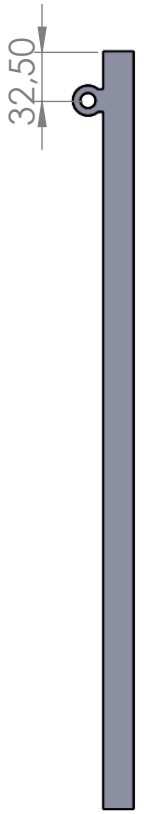


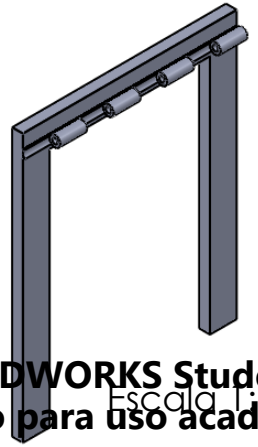


SI NO SE INDICA LO CONTRARIO: LAS COTAS SE EXPRESAN EN MM ACABADO SUPERFICIAL: TOLERANCIAS: LINEAL: ANGULAR:		ACABADO:		REBARBAR Y ROMPER ARISTAS VIVAS		NO CAMBIE LA ESCALA	REVISIÓN
	NOMBRE	FIRMA	FECHA			TÍTULO:	
DIBUJ.	Aitor García Catoira		16/07/17				
VERIF.							
APROB.							
FABR.							
CALID.				MATERIAL: Aluminio		N.º DE DIBUJO Pieza chapa metálica interior 2	A4
				PESO:		ESCALA:1:5	HOJA 4 DE 6





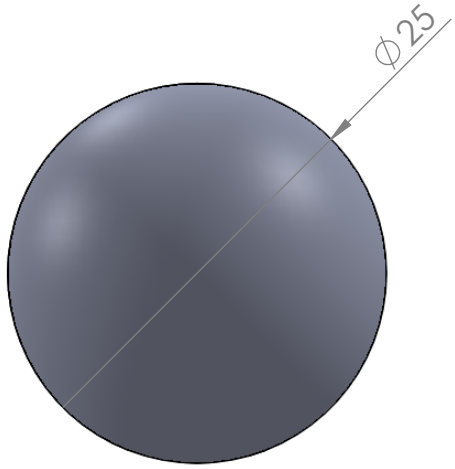




Escala 1:10

SOLIDWORKS Student Edition.
Solo para uso académico.

SI NO SE INDICA LO CONTRARIO: LAS COTAS SE EXPRESAN EN MM ACABADO SUPERFICIAL: TOLERANCIAS: LINEAL: ANGULAR:	ACABADO:		REBARBAR Y ROMPER ARISTAS VIVAS		NO CAMBIE LA ESCALA	REVISIÓN
	NOMBRE	FIRMA	FECHA			TÍTULO:
DIBUJ.	Aitor García Catoira		16/07/17			
VERIF.						
APROB.						
FABR.						
CALID.				MATERIAL: Aisi 316L	N.º DE DIBUJO Caballete	A4
				PESO:	ESCALA:1:5	HOJA 5 DE 6



SI NO SE INDICA LO CONTRARIO: LAS COTAS SE EXPRESAN EN MM ACABADO SUPERFICIAL: TOLERANCIAS: LINEAL: ANGULAR:		ACABADO:		REBARBAR Y ROMPER ARISTAS VIVAS		NO CAMBIE LA ESCALA	REVISIÓN
	NOMBRE	FIRMA	FECHA			TÍTULO:	
DIBUJ.	Aitor García Catoira		16/07/17				
VERIF.							
APROB.							
FABR.							
CALID.				MATERIAL: Indefinido		N.º DE DIBUJO Esfera del puntero	A4
				PESO:		ESCALA:1:5	HOJA 6 DE 6