Universidad Zaragoza
1542

Trabajo Fin de Grado

Simulación háptica en tiempo real de contacto entre sólidos deformables.

Autor

Adrián Berges Enfedaque

Directores

Icíar Alfaro Ruiz
Carlos Quesada Granja

Escuela de Ingeniería y Arquitectura de la Universidad de Zaragoza

Año 2014/2015

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

D./Dª. ! ŘNƛłƴ . ŜNƎŜă 9ƴŧŜŘlꞏljʤzŜ,

con nº de DNI TOЛMCMTП, en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) DNlꞏŘ2, (Título del Trabajo) {ƛƳʤzflꞏǑƛƽƴƘłLJűƛǑlꞏ ŜƴƘűƛŜƳLJ2 NŜlꞏf ŘŜ Ǒ2ƴűlꞏǑű2 ŜƴűNŜ ăƽfƛŘ2ă ŘŜŧ2NƳlꞏōfŜăΦ

_____

_____

_____

_____

_____,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, нc ŘŜ Wdzƴƛ2 ŘŜ нлмp

Fdo: ! ŘNƛłƴ . ŜNƎŜă 9ƴŧŜŘlꞏljʤzŜ

AGRADECIMIENTOS

A mi familia y amigos, por el apoyo que han supuesto siempre en mi vida.

A mis compañeros de carrera, con los que pasé cuatro años trabajando codo con codo para convertirnos en ingenieros.

Y en especial, a mis directores, Carlos Quesada e Icíar Alfaro, por su inestimable ayuda y por lo mucho, muchísimo que he aprendido con ellos.

TABLA DE CONTENIDOS

ÍNDICE DE FIGURAS

1

INTRODUCCIÓN

1 RESUMEN

El contexto en el que se desarrolla el presente Trabajo de Fin de Grado es un proyecto para desarrollar un simulador de cirugía general por parte del grupo AMB (*Applied Mechanics and Bioengineering*) del Departamento de Ingeniería Mecánica de la Universidad de Zaragoza.

El desarrollo de simuladores de cirugía general tiene un especial interés en Medicina, no solo por su aplicación en labores docentes (entrenamiento de los futuros cirujanos), sino también para la planificación y ensayo de operaciones que aumenta la seguridad del paciente. Los simuladores pueden clasificarse en dos grandes categorías: simuladores mecánicos y simuladores virtuales.

Los simuladores mecánicos trabajan sobre un modelo físico del problema. Por ejemplo, en la Figura 1. Ejemplos de simulador mecánico. puede verse un simulador consistente en una pantalla, una mesa donde se colocará el modelo sobre el que se desarrolla la operación y dos instrumentos para manipular el modelo.

Figura 1. Ejemplos de simulador mecánico.

Estos simuladores pueden ser suficientemente realistas en muchas aplicaciones, y la respuesta táctil del modelo es inmediata al tratarse de una construcción sólida; pero plantean

dos limitaciones: la primera, el material debe ser, como se ha dicho, suficientemente parecido al tejido humano. La segunda es que, la complejidad del modelo viene limitada por los medios de fabricación disponibles en la actualidad.

Por otro lado, los simuladores virtuales como el que desarrolla el AMB tienen igualmente una pantalla e instrumentos de manipulación, pero el modelo mecánico se ve sustituido por un modelo virtual. Las limitaciones de esta familia de simuladores vienen dadas por la potencia de cálculo disponible y la eficiencia de los algoritmos usados.

Uno de las principales retos de los simuladores virtuales es obtener una respuesta háptica (es decir, táctil). Para ello se usa normalmente un dispositivo con servomotores (brazo háptico) que el usuario utiliza como entrada al programa, y a su vez el dispositivo devuelve al usuario las fuerzas calculadas. Sin embargo, así como el sentido de la vista puede percibir movimiento fluido con frecuencias de actualización de 25 Hz, el tacto es más sensible y es necesario alcanzar los 1000 Hz, lo cual aumenta la velocidad de cálculo requerida por el simulador.

Además, surge el problema, ubicuo en el mundo de la computación, de la detección de colisión entre sólidos; algo necesario si se pretende lograr un simulador capaz de trabajar con más de un órgano a la vez.

Para ilustrar la situación actual, se muestra el sistema para clasificar las capacidades de un simulador de cirugía (originalmente dada por R. Satava), que define 5 generaciones [1]:

GENERACIÓN I	Representación precisa de la geometría de los órganos a nivel macroscópico.
GENERACIÓN II	Simulación realista de la dinámica de los tejidos en tiempo real. Se incluye respuesta háptica.
GENERACIÓN III	Se incluye la capacidad fisiológica del órgano en el modelo. Implica simular los mecanismos que regulan el funcionamiento del órgano.
GENERACIÓN IV	Anatomía microscópica, es decir, el modelo incluye vasos sanguíneos y sistema nervioso a nivel microscópico (capilares y terminaciones nerviosas respectivamente).
GENERACIÓN V	Descripción del sistema biológico a nivel bioquímico.

Hoy en día nos encontramos con simuladores que pertenecen a la generación VI, aunque en realidad ninguno llega a representar el comportamiento del órgano de una forma suficientemente realista.

Por todo ello, se hacen necesarias nuevas metodologías de resolución numérica de sistemas de ecuaciones diferenciales, que junto a un diseño inteligente del software, permita cumplir el reto de simular a 500 Hz un sistema de varios sólidos deformables no lineales, que interactúen entre sí y con el usuario de forma visual y háptica – el reto de conseguir un simulador de cirugía virtual de generación II completamente funcional.

El presente Trabajo de Fin de Grado se ocupará del postprocesado de una solución mediante el método de la Descomposición Propia Generalizada para dos sólidos elástico-lineales, logrando la representación visual y háptica de su colisión, de forma interactiva con un usuario (esto es, el usuario usará un dispositivo háptico para interactuar con los sólidos, y si la deformación de uno de ellos le llevase a colisionar con el otro, se representará dicha colisión).

2 OBJETIVO Y ALCANCE DEL PROYECTO

La labor realizada el presente Trabajo de Fin de Grado es el desarrollo de un programa para la simulación en tiempo real de la colisión entre dos sólidos deformables.

El propósito del Trabajo de Fin de Grado es probar que es posible alcanzar una frecuencia de actualización del estado de la simulación de 500 a 1000 Hz para el caso de dos sólidos deformables que pueden interactuar entre ellos y con el usuario.

Para ello, se ha desarrollado una aplicación con el lenguaje de programación C++ en entorno Microsoft Visual Studio 2010 encargada de realizar la simulación, haciendo uso de las API[1] de OpenGL y OpenHaptics, siendo una parte fundamental de la simulación lograr una respuesta háptica (es decir, táctil) a través de un dispositivo *PHANTOM OMNI* de la marca Sensable™. Se partió del código de una aplicación anterior que calculaba las deformaciones de un sólido.

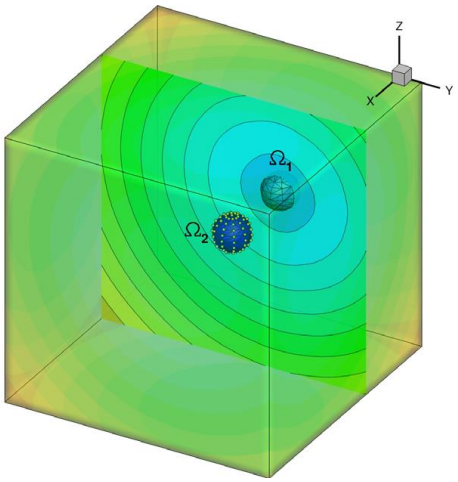


Figura 2. Ejemplo de un distance field (Ω_1) y pointshell (Ω_2).

La simulación se basa en la obtención en tiempo real de soluciones a partir de un problema resuelto mediante el método de reducción de modelos conocido como Descomposición Propia Generalizada (PGD por sus siglas en inglés, Proper Generalized Decomposition). Para lograr la colisión, se hizo uso de técnicas de *distance field*. La combinación de PGD con *distance field* resulta fundamental para lograr las frecuencias de cálculo necesarias.

Como parte del proyecto se desarrolló un segundo programa, encargado de hacer el cálculo de la discretización del *distance field* usado para la gestión de la colisión.

3 ESTRUCTURACIÓN DEL DOCUMENTO

A lo largo de la memoria de este Trabajo de Fin de Grado, el lector podrá encontrar:

- Una introducción a la noción de *distance field* y su uso en este Trabajo de Fin de Grado.

[1]Acrónimo inglés para *Application Programming Interface*, designa al conjunto de métodos de una librería para ser utilizado por otro software como una capa de abstracción.

- Una introducción al método PGD de resolución de ecuaciones diferenciales en derivadas parciales.
- Organización general de las aplicaciones, describiendo brevemente su diagrama de flujo.
- Descripción de los algoritmos utilizados en el programa.
- Resultados de la simulación de la colisión.

Los capítulos que siguen a este primero, que sirve de introducción, desarrollarán los siguientes aspectos:

CAPÍTULO II. Aspectos teóricos del *distance field* y su aplicación en el presente TFG, antecedentes.

CAPÍTULO III. Aspectos teóricos de la PGD, antecedentes y estado del arte, ejemplo de problema resuelto mediante PGD.

CAPÍTULO IV. Dedicado a explicar el funcionamiento del programa de simulación.

CAPÍTULO V. Dedicado a explicar el funcionamiento del programa de cálculo del *Distance field*.

CAPÍTULO VI. Resultados.

CAPÍTULO VII. Conclusiones y Líneas Futuras.

ANEXO A. Contiene un ejemplo de solución de problema mediante PGD.

ANEXO B. Código fuente de la aplicación dedicada a la simulación.

ANEXO C. Código fuente del programa usado para el cálculo del *distance field*.

ANEXO D. Información sobre recursos utilizados: brazo háptico, OpenGL y OpenHaptics.

2

DISTANCE FIELDS Y POINTSHELLS

1 INTRODUCCIÓN

Intuitivamente, parece claro que para cualquier superficie S en el espacio euclídeo, puede definirse una función

$$f(x_1, x_2, x_3) = d$$

tal que d es la distancia más corta (es decir, la euclídea) a la superficie S. Si dicha superficie es cerrada, también se puede definir la función de forma que tome valores negativos en la región del espacio que queda delimitada por S, y positivos en el resto. Podría llamarse a esta función el campo de distancias asociado a S.

Precisamente, lo que en computación se conoce como *distance field* (campo de distancias) no es más que una discretización de la función continua anteriormente descrita. Es decir, se toma una muestra de puntos del espacio, y se evalúa la función *f* en dichos puntos, creando una estructura de datos formada por elementos de la forma:

$$\big(x_1, x_2, x_3, f(x_1, x_2, x_3)\big)$$

El tener una variable conteniendo información de la distancia a una superficie permite ahorrar una gran cantidad de cálculos de distancia euclídea, y es un recurso muy útil en simulación háptica de sólidos con geometría no trivial.

Si se quisiera simular únicamente el contacto entre una herramienta quirúrgica (simulada como el extremo del brazo háptico) y un sólido, bastaría con evaluar la posición dicho extremo para comprobar si está dentro o fuera del sólido. Sin embargo, en la colisión entre dos sólidos surge la cuestión de qué nodos van a ser evaluados dentro del *distance field*. Dichos nodos se conocen como *pointshell*, la cual consiste en una malla de puntos superpuesta a la malla modelo del objeto, cada uno con una normal asociada que apunta hacia adentro. Dichos nodos pueden o no coincidir con los de la malla, siendo el caso más simple aquel en el que el *pointshell* está compuesto por todos los nodos de la malla modelo.

Para su uso en la colisión, se comprueba para cada ciclo háptico si los puntos del *pointshell* están en el *distance field*, y se obtiene la distancia para aquellos que estén, lo que detectaría la colisión en caso de obtener distancia negativa para alguno de ellos.

Finalmente, se define el concepto de voxel (combinación de las palabras volumen y pixel), que se debe interpretar como el equivalente tridimensional de un pixel: sencillamente es un

hexaedro regular, pero su función es, análogamente a un pixel en una imagen bidimensional, servir como bloque básico de construcción de un volumen. Por tanto, todos los voxeles deben ser iguales. En este contexto, un voxel equivale a un "elemento" de la malla del *distance field*.

2 ANTECEDENTES

El uso de un *pointshell* es anterior al uso de un *distance field* en colisión háptica. Como se ve en [2], el nombre dado a la técnica para lograr el contacto entre sólidos rígidos es *Voxmap-PointShell* (VPS). Para resumir dicho enfoque, los voxeles tenían asociado un campo de 2 bits (4 valores posibles), que codificaba los posibles estados del volumen: espacio libre, interior del sólido, superficie y proximidad. Esto conformaba el llamado *Voxmap*.

Bajo este modelo, cuando un nodo del *pointshell* es detectado en un volumen del interior (con un cierto offset para evitar interpenetración), se define un plano que pasa por el centro del voxel y tiene la dirección de la normal. La distancia entre el nodo y el plano se interpreta como la distancia de penetración, y un modelo de muelle-amortiguador calcula la fuerza que se debe devolver.

Este enfoque implica estar calculando las distancias en tiempo de ejecución, siendo el *voxmap* usado de forma puramente cualitativa.

Posteriormente, J. Barbič y D. L. James usan el enfoque VPS para lograr la simulación de sólidos deformables [3], cambiando en este caso el campo de 2 bits por valores de punto flotante con signo, es decir, distancias precalculadas: un *distance field* que sustituye al anterior *voxmap*. En esta implementación, se indexan los puntos del *pointshell* según un árbol jerárquico, de forma que se pueda comprobar la penetración progresivamente, y cortar la comprobación de puntos para lograr la velocidad de computación necesaria en un entorno háptico, a costa de perder exactitud.

La fuerza en este caso es calculada con fuerzas de "penalización", de la forma:

$$\boldsymbol{F} = -k_C d\boldsymbol{N},$$

donde k_C es la rigidez de la fuerza de contacto, de valor arbitrario, $d < 0$ es la distancia de penetración (obtenida directamente del *distance field*), y $\boldsymbol{N}$ es la normal del nodo del *pointshell*. En este modelo, la dirección de la fuerza está determinada únicamente por el sólido que tiene el *pointshell*. Esta aproximación ha sido usada en trabajos más recientes como [4]. La aplicación aquí desarrollada calcula, en cambio, una fuerza proporcional a la deformación obtenida por PGD.

Finalmente, en la colisión de sólidos deformables, el *pointshell* deberá ser deformable, y en general también lo tendrá que ser el campo de distancias. Como se explica en el capítulo 5 de esta memoria, en este Trabajo de Fin de Grado se usa un *pointshell* deformable, pero no un *distance field* deformable a pesar de tener dos sólidos deformables.

3

DESCOMPOSICIÓN PROPIA GENERALIZADA

1 INTRODUCCIÓN

En el resumen que abría el presente Trabajo de Fin de Grado se habló de la necesidad de desarrollar nuevos métodos de cálculo para superar los retos que plantea un simulador de cirugía virtual de segunda generación. En general, lo mismo puede decirse de varios grupos de problemas de computación:

- Modelos de alto número de dimensiones, en los cuales, los métodos tradicionales basados en malla alcanzan tamaños de malla completamente inabordables por cualquier ordenador disponible actualmente, ya que el número de ecuaciones del sistema algebraico a resolver es M^D, siendo M el número de nodos por eje y D el número de dimensiones (grados de libertad del problema).
- Modelos paramétricos, en los cuales dichos parámetros deben suponerse o medirse. Un modelo paramétrico puede, en la práctica, asumirse como un modelo multidimensional si se trata cada parámetro como una dimensión extra.
- Aplicaciones de datos dinámicos (DDDAS, Dynamic Data-Driven Application Systems), en las cuales, las condiciones de frontera del problema varían durante la simulación (por ejemplo en un simulador de cirugía, variará el nodo en el que se aplica la carga en el sólido según mueva el dispositivo el usuario). La variación de los datos puede asumirse como parámetros de la simulación.

Es evidente que los tres grupos mencionados están muy relacionados, por lo que es esperable que la misma estrategia pueda servir para todos ellos, con mínimas variaciones.

Sin embargo, atacar dichos problemas por fuerza bruta simplemente no es una opción. Para entenderlo, podría ponerse de ejemplo un modelo con 30 grados de libertad, algo que se considera incluso simple [5], y una malla grosera de 1000 nodos por dimensión. La malla resultante contiene 10^{90} nodos. El superordenador más rápido que existe en la actualidad (el *Tianhe-2*, situado en el Centro Nacional de Supercomputación, Guangzhou, China) tiene 3 millones de procesadores y es capaz de una velocidad de cálculo de 33.86 PFLOP/s. Incluso aunque pudiéramos resolver un nodo de la malla en una única operación de punto flotante (FLOP), el tiempo de cálculo sería:

$$\frac{10^{90}\ [\text{FLOP}]}{33.86 \cdot 10^{15}\ [\text{FLOP/s}]} = 2.95 \cdot 10^{73}\ \text{s}$$

El tiempo total de vida del Universo desde el Big Bang hasta su hipotético final se estima en $4.32 \cdot 10^{34}$ s. Serían necesarias las vidas de varios sextillones (!) de universos como el nuestro para obtener la solución (y eso bajo el supuesto de resolver un nodo con una sola operación, algo que en la práctica no se da, ni mucho menos).

Resulta claro que la clave para solventar estas dificultades está en el desarrollo de metodologías más eficientes.

2 ANTECEDENTES

Si se presta atención a la historia de la ingeniería, se verá cómo ante la escasez de recursos computacionales, siempre se ha recurrido a la reducción de modelos. En efecto, aun hoy, el modelo de barra unidimensional, en el que se calcula un sólido tridimensional utilizando como variables el desplazamiento y giro de la directriz, está presente en muchos cálculos estructurales.

El ejemplo de la barra unidimensional ilustra bien la filosofía de la reducción de modelos: "extraer" la mayor cantidad de información de la solución mediante el modelo más sencillo posible.

Los solucionadores generalizados que se han desarrollado en la mayor parte de los campos científicos han venido a llamarse Proper Orthogonal Decomposition (POD) o también Principal Component Analysis (PCA) entre otros muchos nombres. El objetivo de este tipo de métodos es obtener (resolviendo problemas similares al que uno quiere resolver) una serie de funciones que contengan la mayor información posible de la solución. Se confía en que dichas funciones encontradas no difieran mucho de la solución exacta del problema.

En general, lo que se pretende en POD/PCA, para un problema de D dimensiones, es obtener una aproximación del tipo:

$$u(\overline{x}) \approx \sum_{i=1}^{N} \phi_i^1(x_1) \cdot \phi_i^2(x_2) \cdot \cdots \cdot \phi_i^D(x_D) = \sum_{i=1}^{N} \prod_{k=1}^{D} \phi_i^k(x_k)$$

Donde $\phi_i^k(x_k)$ son funciones dependientes únicamente de la variable correspondiente. Es importante notar que pueden agruparse variables en vectores, si resulta conveniente. Es decir, no es necesaria una separación total de las variables.

El algoritmo encargado de hacer el análisis POD/PCA calcula las funciones como base de un espacio funcional, de modo que los vectores (funciones) estén ordenados de mayor a menor según alguna norma definida sobre el espacio (normalmente en problemas de física será la energía).

La solución de casos particulares del problema multidimensional formulado, denominados *snapshots*, puede hacerse mediante algún método de solución numérica exacta como FEM o pueden ser, sencillamente, resultados experimentales. De dichas soluciones similares se extraen los vectores propios, que van conformando un espacio funcional, ordenado de mayor a menor valor propio.

Para intentar solventar la necesidad de solución a priori de snapshots, P. Ladeveze desarrolló un método denominado como *LArge Time INcrements* (LATIN) que no necesita de inspección de soluciones para construir la reducción del modelo [6] [7]. En concreto, fue desarrollado para la descomposición de la solución en dominios de espacio y de tiempo (estando las tres coordenadas espaciales agrupadas en un vector).

Más recientemente, F. Chinesta generalizó de manera independiente esta aproximación, orientándola al problema multidimensional (parametrización), obteniendo así el método Proper Generalized Descomposition (PGD), que es la técnica usada en este proyecto.

3 LA DESCOMPOSICIÓN PROPIA GENERALIZADA

La metodología de la PGD es una generalización de POD/PCA, y al igual que en dicho método, se busca obtener una forma separada de la solución:

$$u(\overline{x}) \approx \sum_{i=1}^{N} X_i^1(x_1) \cdot X_i^2(x_2) \cdot \cdots \cdot X_i^D(x_D) = \sum_{i=1}^{N} \prod_{k=1}^{D} X_i^k(x_k) \qquad (3.1)$$

Sin embargo, se difiere del método anterior en que no se resuelve a priori ningún *snapshot*, sino que las bases se calculan sobre la marcha.

Puede comprobarse que, en esta forma, parámetros del problema (p_i^j) que puedan tener un rango continuo de valores se pueden tratar como variables del problema:

$$u(\overline{x}, p_1, \dots, p_S) \approx \sum_{i=1}^{N} \left(\prod_{k=1}^{D} X_i^k(x_k) \cdot \prod_{j=1}^{S} P_i^j(p_j) \right) = \sum_{i=1}^{N} \prod_{k=1}^{D'} X_i^k(x_k)$$

Donde $D' = D + S$ y los distintos p_j se han asumido como nuevas variables y por tanto renombrados como x_k.

Como se dijo anteriormente, las variables no tienen por qué ser escalares, sino que se pueden agrupar en vectores, siendo por tanto algunas funciones dependientes de varias variables. Para formalizar esto, sea el hiperdominio Ω en el que está definido el problema, la separación de variables que más convenga se traduce en una división del dominio tal que: $\Omega = \Omega_1 \times \cdots \times \Omega_M$, siendo cada subdominio de dimensión D_i y por tanto $D = \dim(\Omega) = \dim(\Omega_1) + \cdots + \dim(\Omega_M) = D_1 + \cdots + D_M = \sum_{i=1}^{M} D_i$. Al separar el dominio, se reduce el número de soluciones de M^D a $N \cdot M \cdot D$ [8], clave para conseguir la reducción de tiempo de cálculo necesaria en este tipo de problemas.

4 UTILIZACIÓN DE PGD PARA LA SIMULACIÓN EN TIEMPO REAL DE COLISIONES ENTRE SÓLIDOS

El simulador desarrollado en este trabajo de fin de grado utiliza como datos de entrada soluciones calculadas mediante PGD.

Cuando dos sólidos contactan, en sus superficies aparecen fuerzas de contacto que deforman los sólidos. Como el contacto puede ser en cualquier posición, o en cualquier región, en el planteamiento PGD del problema se ha introducido como parámetro la posición de esas fuerzas. Se propone una separación en la que el desplazamiento depende del punto considerado y de la posición del contacto o, equivalentemente, la posición de la fuerza de contacto, denominada *s*.

$$\boldsymbol{u}(\overline{x}) = \begin{pmatrix} u_x \\ u_y \\ u_z \end{pmatrix} \approx \begin{pmatrix} \sum_{i=1}^{N} F_x(x,y,z) \cdot F_s(s) \\ \sum_{i=1}^{N} F_y(x,y,z) \cdot F_s(s) \\ \sum_{i=1}^{N} F_z(x,y,z) \cdot F_s(s) \end{pmatrix}$$

La ventaja de usar este método es que una vez resuelto el problema se obtiene un vademecum con la información del desplazamiento de cualquier punto para cualquier posición de carga separada en dos funciones. En la simulación del contacto hay que detectar qué puntos están en contacto (coordenadas *s*) y calcular, mediante la ecuación anterior, la deformada correspondiente a cada sólido y los valores de las distintas fuerzas que aparecen en el sistema. Todos estos cálculos pueden hacerse en tiempo real.

En el Anexo A se detalla la formulación PGD para el problema estático de elasticidad lineal, que es el más sencillo. El grupo de investigación AMB ha desarrollado soluciones PGD en la que separan el espacio y la posición de la carga para problemas estáticos y dinámicos, y planteamientos lineales y no lineales. El código de programación desarrollado en este trabajo puede utilizarse para simular la colisión de cualquiera de los problemas estáticos formulados por el grupo AMB.

4

PROGRAMA SIMULADOR

1 INTRODUCCIÓN

Dado que la tarea principal del TFG ha sido el desarrollo de las aplicaciones, el presente capítulo es especialmente importante, ya que en él se detalla el funcionamiento de las mismas. En este apartado se dará un repaso a la evolución del código durante el TFG, sin pretender dar los detalles del código anterior (que no tiene sentido explicar ya) ni del código de la implementación final (para lo cual se usan los otros apartados del capítulo)

Inicialmente, el diseño del programa era imperativo, esto es, un único fichero contenía todas las funciones[2] y variables necesarias para hacer la simulación. Los problemas de este tipo de enfoque eran:

- Fichero de una extensión enorme, difícil de navegar.
- Poca abstracción de los procesos. Como consecuencia, era fácil incurrir en soluciones *ad hoc* que restaban reusabilidad al código.
- En general resultaba complicado entender qué hacía el código y la corrección de errores podía volverse muy costosa en tiempo.

Por ello, en seguida se pasó a un enfoque más moderno, basado en el paradigma de la programación orientada a objetos (OOP por sus siglas en inglés). En ella, la funcionalidad del código se separa en diversas clases que se encargan de unas pocas tareas. Esto hace que sea mucho más fácil la abstracción de procesos, y que sea más fácil entender el funcionamiento del programa en un vistazo.

La relación entre ellas se puede ver en el diagrama de la Figura 3. Arbol de clases de la simulación..

Como `c++` exige una función `main`, se usa el archivo `Programa` para contener esa función y ser el nexo de unión del resto de clases (por así decirlo, en `Programa` se define la escena a simular). También cuenta con gran parte de las funciones gráficas y hápticas, así como la gestión de la interacción con el usuario.

[2] En el sentido informático, es decir, rutina o método.

Hay que tener en cuenta que el estudiante es Ingeniero Mecánico, no Informático, y por tanto sus conocimientos en materia de diseño de software fueron adquiridos durante el proyecto. Con ello se quiere decir que el diseño del programa todavía puede ser mejorado (sin ir más lejos, la clase `Solid` tiene una extensión de 365 líneas, contrastando con la mayoría de clases usadas en el programa que tienen extensiones en torno a las 100 líneas, lo que indica que el código no es suficientemente abstracto y podría ser separada en clases más pequeñas y de menor extensión).

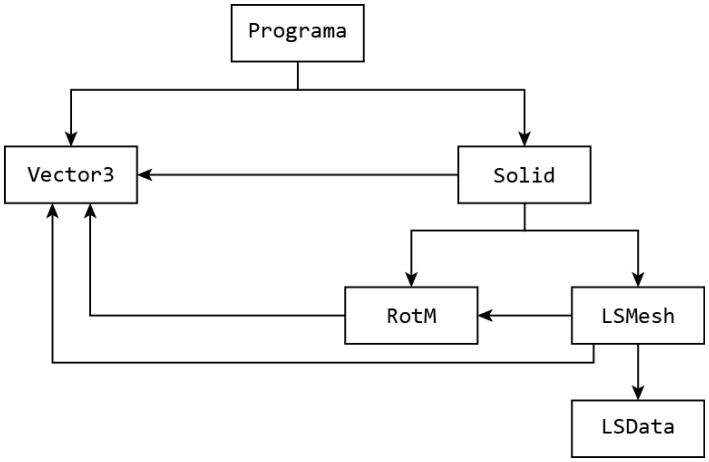


Figura 3.
Arbol de clases de la simulación. La clase que recibe la flecha es usada por la clase de la que origina la flecha.

2 FICHEROS DE ENTRADA

La solución del problema mediante PGD se almacena en cuatro ficheros (uno para cada variable, ya que en esta simulación la solución depende de las tres variables espaciales y una extra para el parámetro de la posición de la carga): `Fx.h`, `Fy.h`, `Fz.h`, `Fs.h`.

El fichero `malla.h` contiene datos sobre la malla del modelo tridimensional del sólido deformable.

En `parametros.h` se guarda información acerca de la solución, tal como el número de modos de la PGD o el valor de la carga puntual que se aplicaba sobre los nodos.

La salida del programa que calcula el *distance field*, el fichero `LSData.h`, contiene la malla del mismo, así como las distancias calculadas.

2.1 Malla

Los datos de la malla del sólido están almacenados en forma de matrices. Se hace una distinción entre la superficie háptica[3] y la superficie total del sólido, donde la matriz `CoorSup` es la que contiene las coordenadas de cada vértice de la superficie total del sólido, y `CoorS` la

[3] La superficie háptica es la región que puede tocarse con el brazo háptico o contactar con otros sólidos, es decir, la región en la que puede estar aplicada la carga. Esa región puede coincidir con la superficie total o ser una región de la misma.

que contiene las coordenadas de los vértices de la superficie háptica. En general la nomenclatura del programa usa el sufijo "Sup" para la superficie total y "S" para la háptica.

Esta separación es para evitar el renderizado en OpenHaptics de toda la superficie del sólido, algo costoso y no muy útil pues sólo es necesario representar en el espacio háptico los nodos parametrizados en la solución de PGD (y que por tanto son tocables durante la simulación).

Las dos matrices `ConnectSup` y `ConnectS` definen los triángulos de la superficie. Contienen ternas de números enteros, que son índices para la matriz `Coor` correspondiente. Por ejemplo, `{1, 2, 57}` significa que el primer triángulo está formado por los vértices 1, 2 y 57 (el orden influye en el sentido de la normal, lo cual es muy importante ya que OpenGL tiene una opción para pintar únicamente los triángulos visibles desde la cámara, es decir aquellos cuyas normales estén orientadas hacia la cámara).

`LoadedNodesSup` relaciona los vértices de la superficie total con los de la superficie háptica.

Aunque pueda parecer que se está duplicando información dentro de este fichero, la razón es que tal y como funciona OpenHaptics, es mejor tener la malla de la superficie háptica como una entidad separada, para agilizar el cálculo en el *thread* de alta prioridad que usa la API del brazo. Si pasamos toda la geometría del sólido, podríamos estar renderizando miles de vértices cuando en realidad sólo interesan unas pocas decenas, aquellos que estaban cargados en la solución PGD.

2.2 Ficheros solución PGD y parámetros

En el Capítulo 3 se decía que en la práctica, la solución mediante PGD sobre la malla que discretiza el dominio era un producto de funciones. Los ficheros `Fx.h`, `Fy.h`, `Fz.h` y `Fs.h` contienen cada una de esas matrices.

En `parametros.h` se guardan dos variables, la fuerza usada en el problema paramétrico, y el número de modos del PGD (número de sumandos de la aproximación).

2.3 Fichero de resultados de *Distance field*

El fichero `LSData.h` contiene dos matrices: `LSVertices` contiene las coordenadas de cada nodo de la malla del *Distance field*, vértices de los vóxeles y `LSMatrix` la distancia que hay desde cada uno de los puntos de la malla del distance field hasta el nodo más cercano de la superficie indeformada del sólido a simular.

Para un punto de la matriz de vértices, `LSVertices[i]`, la distancia que corresponde a dicho punto es `LSMatrix[i]`.

3 CLASES AUXILIARES

Se hace uso de dos clases auxiliares, que se encargan de facilitar la manipulación de ciertos datos geométricos (puntos en el espacio y direcciones en el caso de `Vector3`, y rotaciones en el caso de `RotM`).

3.1 Vector3

Esta clase fue creada para la gestión de ternas de números que representen tanto puntos como vectores. Actualmente el código opera casi por completo usando éstos vectores, simplificando mucho la escritura de operaciones típicas de la computación geométrica, como suma de puntos o productos vectoriales.

Es importante notar que esta clase hace uso de la característica de *plantilla* de `C++`, lo que le da la versatilidad de poder ser definido como un vector de `double` (valor que toma por defecto), o de `int`, o `float`, o cualquier otro tipo o clase definido (incluso se podría definir una matriz de 3x3 como un `Vector3<Vector3<…>>`).
Los siguientes operadores han sido sobrecargados para facilitar el uso de la clase:

- `=` Asigna un Vector3 a otro (copia elemento por elemento).
- `+` Suma dos vectores elemento a elemento.
- `-` Resta dos vectores elemento a elemento.
- `*` Producto escalar entre dos vectores.
- `^` Producto vectorial entre dos vectores.
- `+=, -=` Asignación con suma y asignación con resta.
- `==, !=` Operadores lógicos de comparación (“igual a” y “no igual a”).
- `[]` Operador subíndice. Permite acceder a los elementos del vector.

Adicionalmente, se han definido la función `GetModulus`, que devuelve el módulo del vector.

En el Capítulo 7 se recogen algunas ideas de mejora que podrían implementarse en un futuro, aunque la clase ya cumple su función.

3.2 RotM

Esta clase tiene la función de almacenar la matriz de rotación y su inversa. Para ello cuenta con dos variables, `comp` (valores de la matriz de rotación) e `inv` (valores de la matriz de rotación inversa), y las funciones `SetRotation`, `Rotation` e `InvRotation`.

El constructor[4] de la clase inicia una matriz identidad, es decir, no hay giro.

Mediante la función `SetRotation(a, b, c)` donde `a`, `b` y `c` son los ángulos de rotación θ_X, θ_Y y θ_Z respectivamente. La composición de rotaciones es como sigue: primero se rota en torno al eje X, después en torno al eje Y, y finalmente respecto al Z. En esta misma función se calcula la rotación inversa.

Las función `Rotation(Vector3<> v)` multiplica el vector que se le pase como argumento por la matriz de rotación, efectuando el giro. De forma análoga, `InvRotation(Vector3<> v)` realiza la multiplicación por la matriz inversa.

4 CLASE SOLID

La clase `Solid` ha sido desarrollada para gestionar la geometría del sólido, tanto como sólido rígido (es decir, traslación y rotación) como sólido deformable (usando PGD para calcular el campo de desplazamientos). La ventaja de separar el código de la gestión del sólido en una

[4] En informática, se llama *constructor* a la función que se llama cuando se crea la clase, inicializando las variables.

clase es que se pueden crear múltiples instancias de la clase en una misma escena, permitiendo, por ejemplo, colocar dos sólidos en una determinada configuración con mucha facilidad.

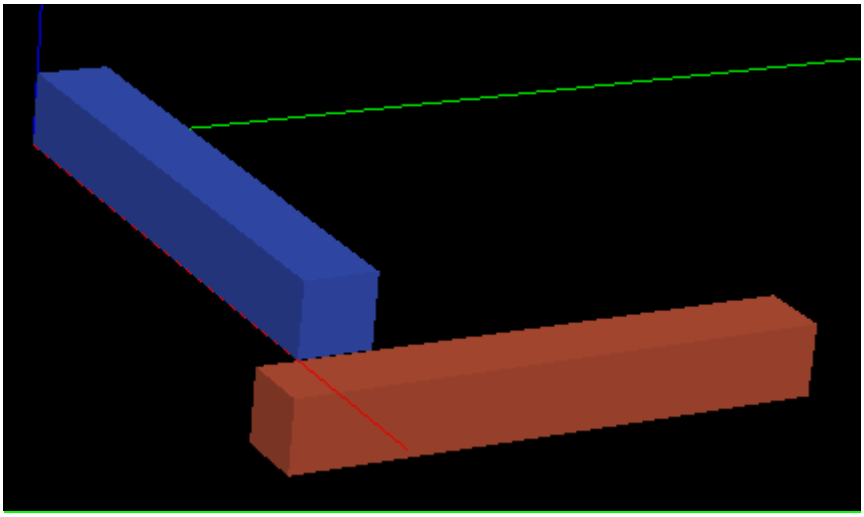


Figura 4. Configuración de los dos sólidos en reposo.

En este apartado se va a hacer hincapié en la función de deformación, aunque se explicarán otras funciones importantes.

4.1 Representación y movimientos como sólido rígido.

Cuando se llama al constructor de la clase, se desarrolla la siguiente llamada de funciones:

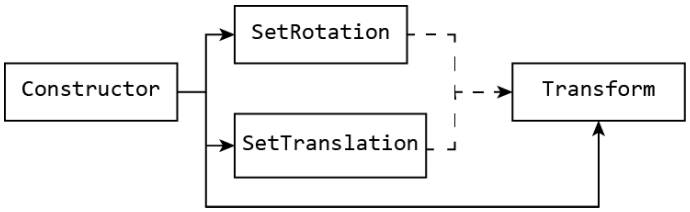


Figura 5. Llamada a funciones en el constructor de la clase Solid

La línea discontinua representa una relación indirecta, ya que `SetRotation` y `SetTranslation` no llaman a ninguna función más, simplemente modifican la matriz de rotación y el vector de traslación.

La función `Transform` recorre todos los vértices y les aplica la matriz de rotación actual y les suma la traslación:

```
_globalCoords[node] = rotation.Rotation(Vector3<>(CoorSup[node][0], CoorSup[node][1], CoorSup[node][2])) + _centerPosition;
```

La variable `_globalCoords[node]` guarda los nodos en las coordenadas globales del espacio de trabajo y es la que se usa para representar el sólido.

Para la representación del sólido, hay que tener en cuenta que se debe generar una superficie poliédrica y luego pasarla a OpenGL o a OpenHaptics. Se ha separado la función que genera la superficie de las funciones para la llamada háptica.

La función `Geometry` se encarga de generar éstos triángulos en `OpenGL`. El funcionamiento del algoritmo está representado en el diagrama de la Figura 6. Algoritmo para generación de la geometría..

Esta función es llamada por las funciones `DrawGL` y `DrawHL`, usadas en el cuerpo del programa principal para representar tanto la escena gráfica como la háptica.

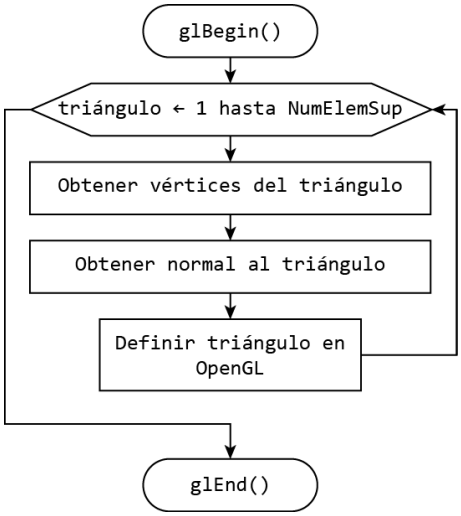


Figura 6. Algoritmo para generación de la geometría.

4.2 Detección del nodo más cercano

Aunque sencilla, la función `GetClosestNode` es clave para todo el proceso, ya que permite conocer, dado un punto cualquiera *P* del espacio, qué nodo de la superficie háptica es el más cercano a él.

Su principal uso en la simulación es averiguar qué nodo usar para calcular los desplazamientos del sólido con la función `Deform`, a partir de la posición del cursor del brazo háptico.

Se puede observar el algoritmo en la Figura 7. Algoritmo de detección de nodo más cercano..

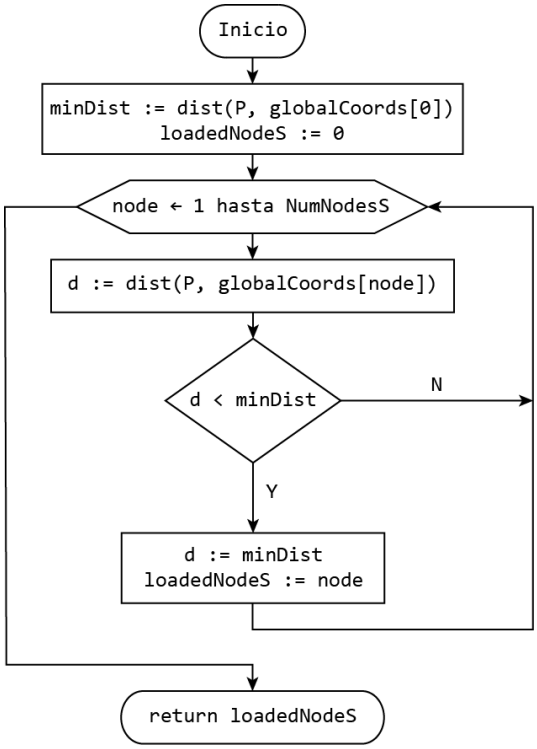


Figura 7. Algoritmo de detección de nodo más cercano.

4.3 Deformación

La función `Deform` calcula el campo de desplazamientos que hay que aplicar a los vértices del sólido indeformado, dado un desplazamiento de uno de los nodos de la superficie háptica.

Para ello, necesita saber el nodo que ha sido cargado (se sabrá gracias a la función `GetClosestNode`) y la distancia en Z recorrida por dicho nodo, desde el estado inicial hasta el que tenga en el *frame* actual (para ello se ha calculado el *distance field*). La Figura 8. Reconstrucción de la solución por PGD para las condiciones de frontera actuales. muestra el diagrama del algoritmo.

La primera tarea es inicializar a cero los desplazamientos (de lo contrario se irían sumando y el cálculo sería erróneo). A continuación se comprueba que la detección del nodo más cercano no dio ningún error.

Los ficheros `Fx.h`, `Fy.h`, `Fz.h` y `Fs.h` tienen almacenada una solución para cada nodo de la superficie háptica del sólido, a la que se le ha aplicado una carga F. Si se reconstruye la solución para una posición de la carga, se obtendrán las deformaciones de los nodos bajo esa hipótesis (se obtiene la viga completamente deformada).

Como en la simulación la deformación puede (y debe) tomar estados intermedios entre el estado indeformado y el completamente deformado, es necesario escalar esos desplazamientos en función del empuje que haya realizado el usuario sobre el sólido mediante el brazo háptico. En este escalado se ha supuesto un comportamiento lineal del problema, asumiendo que cuando el problema sea no lineal se cometerá un cierto error que es asumible en el entorno de la simulación en tiempo real.

Para ello, se obtiene la distancia de penetración respecto a la superficie (dada por el *distance field* tomando como entrada la posición del brazo háptico, ver §4.5) y se divide por el desplazamiento en Z (llamado `RefDisp`) del nodo que se esté tocando en ese momento.

Así se obtiene un factor de carga, `_ScaleFactor`, que se usará tanto para escalar la respuesta háptica como para escalar las deformaciones de los nodos.

Finalmente, una vez se tiene esta solución escalada, se le aplican las transformaciones pertinentes para que esté en coordenadas globales, ya que la reconstrucción de la solución da los desplazamientos en coordenadas locales del sólido.

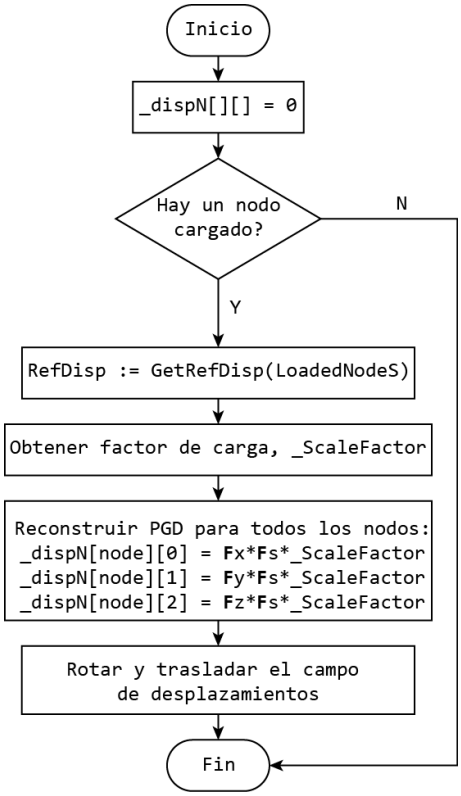


Figura 8. Reconstrucción de la solución por PGD para las condiciones de frontera actuales.

4.4 Cálculo háptico

A través de la variable `_ScaleFactor` se escala la fuerza que ha de devolver el dispositivo.

Esto se hace en el código del archivo `Programa`, usando la función de OpenHaptics `hlEffectd`. A dicha función se le pasa un parámetro definido en la propia librería de OpenHaptics y que indica que el efecto a recrear por el brazo debe ser una fuerza vertical, y un segundo parámetro que es la magnitud de dicha fuerza.

La magnitud es calculada como:

$$|\mathrm{F}| = F_{PGD} \cdot \mathrm{LoadScale}$$

Donde F_{PGD} es la fuerza usada en el cálculo de la solución por PGD, y su valor equivale a 300 N, como se ve en el Capítulo 6. Se encuentra en el fichero `parametros.h`.

5 USO DEL *DISTANCE FIELD*

Una vez se tiene el *distance field* calculado (con el programa descrito en el Capítulo 5), es necesario definir cómo se usa para la gestión de la colisión entre sólidos.

Se desea saber la distancia que tendrá un punto cualquiera del espacio $P \in \mathbb{R}^3$, a la superficie háptica del sólido. Para ello se dispone del *distance field*, el cual contiene puntos de los cuales se sabe la distancia a dicha superficie. El *distance field* está compuesto por hexaedros regulares, esto es, la coordenada en un eje X, Y o Z de un punto difiere de la del punto anterior en una cantidad constante Δx, Δy o Δz respectivamente.

Se tienen 3 sistemas de coordenadas: el del mundo, el del elemento, y el de la matriz del *distance field*.

- (x, y, z) es el sistema de coordenadas global (mundo).
- (i, j, k) es son los índices de la matriz del *distance field*.
- (η, ξ, μ) es el sistema de coordenadas del elemento hexaédrico.

Durante el cálculo, los tres sistemas de coordenadas eran coincidentes. No obstante, durante la ejecución del programa, el *distance field* se orientará y trasladará junto con el sólido al que pertenece, lo que se tendrá que tener en cuenta para hallar el hexaedro en el que está contenido el punto.

5.1 Encontrar los vértices del hexaedro que contiene a un punto

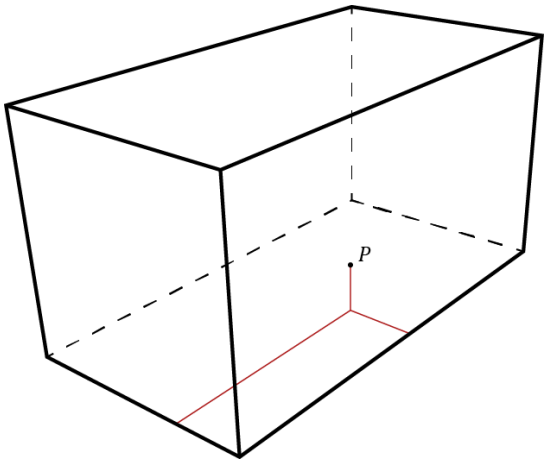


Figura 9.
Punto de $\mathbb{R}^3$ en un hexaedro.

La matriz del *distance field* tiene un ordenamiento interno tal que al recorrer el índice i, se va incrementando la coordenada x de los puntos almacenados (y lo mismo ocurre para j con y y k con z), cuando no hay rotación. Al rotar las coordenadas de los puntos del *distance field*, supóngase $\frac{\pi}{2}$ alrededor del eje Z, dicha relación cambia, y ahora será j el índice que varíe x.

Por ello, se transformará el punto P del espacio (que está en coordenadas globales) al sistema de coordenadas local del *distance field* (que será el mismo que el del sólido). Una vez se tengan las coordenadas de P en este sistema de referencia, se podrán obtener los índices (i, j, k) del punto de la malla del *distance field* (llamado a partir de ahora A) que esté más cercano al origen (punto $(0,0,0)$ en la malla del *distance field*).

En la Figura 10. Designación de los vértices. Si sumamos (i, j k) a cada uno, se obtienen los índices dentro de la malla del Distance field. se muestra la indexación local de un voxel cualquiera del *distance field*. Conociendo los índices (i, j, k) de A, el resto de puntos del voxel se pueden obtener sumando los índices genéricos de la Figura 10. Designación de los vértices. Si sumamos (i, j k) a cada uno, se obtienen los índices dentro de la malla del Distance field. a los de A, definiendo por completo el voxel que encierra al punto P. Aunque esto pueda resultar un poco confuso al principio, resulta más claro para la implementación, ya que de lo contrario el código se vuelve difícil de leer. En cambio, una vez comprendida esta figura, el código es claro.

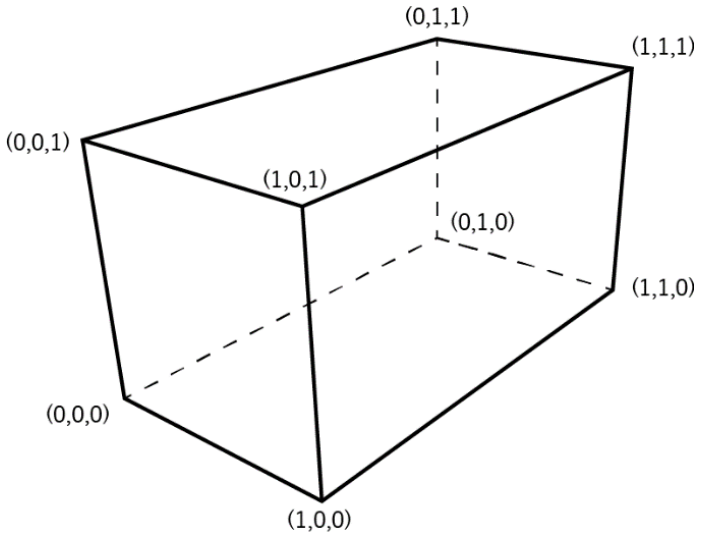


Figura 10.
Designación de los vértices. Si sumamos (i, j k) a cada uno, se obtienen los índices dentro de la malla del Distance field.

Para obtener los índices enteros (i, j, k) de A en la matriz del *distance field*, se debe separar cada una de las tres componentes del punto P del espacio en su parte entera y decimal.

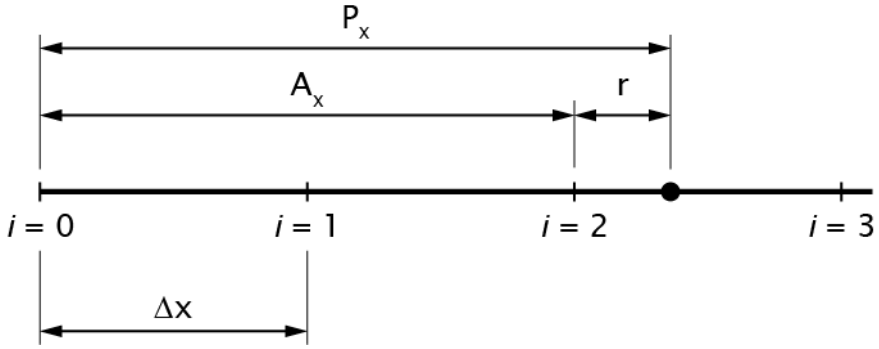


Figura 11. Descomposición de P_x en A_x y r.

Para simplificar la explicación el procedimiento, se va a tomar la componente x de P, y la descomponemos en una suma de dos términos:

$$P_x = \Delta x \cdot i + r = A_x + r$$

De la Figura 11. Descomposición de P_x en A_x y r. se desprende que:

$$\frac{P_x}{\Delta x} = i + \frac{r}{\Delta x}$$

Donde i es la componente homónima del punto A en la matriz del *distance field*.

Evidentemente, no se conoce a priori ni i ni $r/\Delta x$, luego hay que encontrar alguna manera de obtener la parte entera de $P_x/\Delta x$. Afortunadamente, en c++ se puede truncar un número real en su parte entera haciendo una *conversión de tipo*:

```
i = int (P[0] / Dx); //Ahora i es la parte entera de la división.
```

Haciendo lo propio con el resto de componentes de P, se obtendrán j y k.

5.2 Obtención de las coordenadas del punto en el sistema de coordenadas del elemento

El *distance field* sólo contiene la distancia a la superficie para algunos puntos del espacio, por lo que es necesario interpolar los valores en el resto. Esta interpolación se ha decidido hacer con las funciones de forma usadas en FEM, para un elemento hexaédrico lineal de lado dos.

Como se aprecia en la Figura 12. Ejes de coordenadas del elemento., el elemento tiene un sistema de coordenadas local de ejes $\mathrm{H}, \Xi, \mathrm{M}$. Normalmente, para cambiar las coordenadas globales del punto P, del cual se quiere saber la distancia a la superficie, a coordenadas en el elemento se usa la ecuación:

$$\eta = \frac{2x - a - b}{b - a} \qquad (5.1)$$

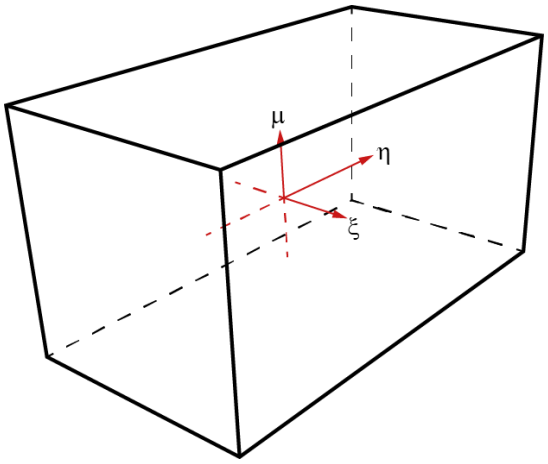


Figura 12.
Ejes de coordenadas del elemento.

Donde a y b designan los extremos del lado del voxel en el eje que corresponda.

Sin embargo, ya que en el paso anterior del algoritmo se obtuvo la parte entera de cada componente del punto del espacio P, se puede aprovechar para tener la parte decimal de la componente normalizada dentro del elemento.

```
double r = (P[0] / Dx) - i; //Elimina la parte entera ya calculada.
```

Ahora, esta coordenada que está entre $[0, 1]$ debe pasar al rango $[-1\,, 1]$ (puesto que el elemento es de lado dos y tiene el origen en el centro). Simplemente se hace la operación:

$$\eta = 2 \cdot (r_x - 0.5) \qquad (5.2)$$

Que, repetido con el resto de componentes del punto, despeja las coordenadas $(\eta,\ \xi,\ \mu)$ de P en el elemento.

Las ecuaciones (5.1) y (5.2) son equivalentes, ya que $a = i \cdot \Delta x$ y $b = (i+1) \cdot \Delta x$, por lo que:

$$\begin{aligned}\frac{2x - a - b}{b - a} &= \frac{2x - i \cdot \Delta x - (i+1) \cdot \Delta x}{(i+1) \cdot \Delta x - i \cdot \Delta x} \\ &= \frac{2x - 2i \cdot \Delta x - \Delta x}{\Delta x} \\ &= 2\left(\frac{x}{\Delta x} - i\right) - 1 \\ &= 2(r_x - 0.5)\end{aligned}$$

5.3 Distancia

Una vez conocidas las coordenadas $(\eta,\ \xi,\ \mu)$ del punto del espacio P, se usan las funciones de forma mencionadas en el apartado anterior para obtener una aproximación de la distancia al punto de la superficie S. En un elemento tridimensional, la función de forma en cada nodo del elemento $N_i^{(e)}$ es [9]:

$$N_i^{(e)} = \tfrac{1}{8}(1 + \eta\eta_i)(1 + \mu\mu_i)(1 + \xi\xi_i) \qquad (5.3)$$

donde:

Nodo	η_i	ξ_i	μ_i
(0,0,0)	-1	-1	-1
(1,0,0)	$+1$	-1	-1
(1,1,0)	$+1$	$+1$	-1
(0,1,0)	-1	$+1$	-1
(0,0,1)	-1	-1	$+1$
(1,0,1)	$+1$	-1	$+1$
(1,1,1)	$+1$	$+1$	$+1$
(0,1,1)	-1	$+1$	$+1$

La distancia de P a S será entonces la suma de los valores ponderados en cada nodo del elemento:

$$d = \sum_{i=1}^{8} d_i N_i$$

donde d_i es el valor de distancia correspondiente al i-ésimo vértice del voxel, y N_i el valor que toma la función de forma en dicho vértice, calculado según la Ec. (5.3).

5

PROGRAMA PARA EL CÁLCULO DEL *DISTANCE FIELD*

1 INTRODUCCIÓN

La información que se debe almacenar en la matriz del *distance field* corresponde a la distancia del punto P de la matriz, a la superficie del sólido deformable.

Por tanto, la función principal del programa es obtener dichas distancias, además de poder determinar si la distancia corresponde a un punto interior al sólido o a uno exterior, de manera que en la fase de simulación sea fácil determinar si un punto cualquiera del espacio ha entrado en el volumen del sólido.

Adicionalmente, debe ser capaz de escribir un fichero de salida con el formato adecuado, para que la simulación pueda usar la información obtenida en este programa.

La idea en la que se basa el algoritmo es la siguiente:

Supongamos una superficie cerrada S diferenciable en todo punto (de modo que sabemos la normal a la superficie en cada punto) y un punto P en el espacio euclídeo $\mathbb{R}^3$.

Entonces, si llamamos $s \in S$ a la proyección ortogonal de P en S, y $\boldsymbol{v}$ al vector que une P con s, siendo $\boldsymbol{n}$ el vector normal a S en s; tenemos que el producto escalar $\boldsymbol{v} \cdot \boldsymbol{n}$ nos da la distancia de P a la superficie. Además, si su signo es negativo, el punto está fuera.

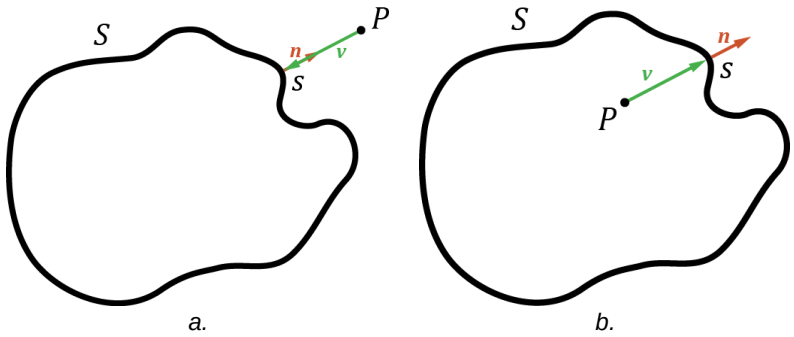


Figura 13.
En el caso (a.), tenemos el punto fuera. Como podemos ver, $\boldsymbol{n}$ y $\boldsymbol{v}$ tienen la misma dirección pero sentido opuesto. En el caso (b.), el punto está dentro, $\boldsymbol{n}$ y $\boldsymbol{v}$ tienen el mismo sentido.

Esto mismo, trasladado a una superficie poliédrica como la que tenemos, la aproximación a la proyección ortogonal es obtener el vértice más cercano.

Para todo ello, el programa hace uso de algunas clases directamente relacionadas con el programa de simulación.

La relación entre las clases usadas puede verse en la figura siguiente:

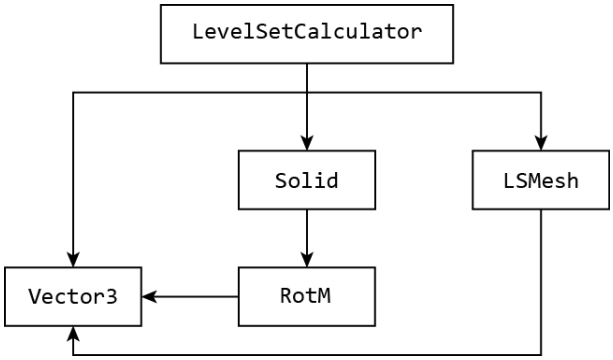


Figura 14. Relación de clases del programa de cálculo del distance field

El fichero `LevelSetCalculator` es el que contiene la función `main`, además de las funciones de cálculo de distancia, detección dentro-fuera, y escritura del fichero de salida. Las clases incluidas son necesarias para que el archivo principal pueda realizar su tarea y, en algunos casos, pueden ser diferentes de las clases con el mismo nombre en el programa dedicado a la simulación. Dichas diferencias se explicarán en el apartado que resulte necesario.

2 CLASE LSMESH

El papel de la clase `LSMesh` es definir una matriz del tamaño necesario para contener todos los puntos del *distance field* y almacenar las distancias que se vayan calculando, para su posterior acceso en la función de escritura del fichero de salida.

Debido a su función, tiene importantes diferencias con la clase `LSMesh` usada en la simulación: obviamente, no incluye funciones para interpolación de la distancia, ya que sólo es necesario en la simulación, cuando ya está calculado el *distance field*. No se puede rotar, ya que tampoco lo va a hacer el sólido en este caso; aunque sí puede trasladarse para ajustarlo a la malla del sólido.

Sin embargo, la diferencia más notable está en los campos que contiene. Esta versión de `LSMesh` tiene tres variables para definir el espaciado de los puntos en x, y y z, otras tres para definir el número de puntos por eje, y finalmente un `vector` (en el sentido de la *standard library* de `C++`) de tipo `double` que se encarga de la gestión dinámica de la memoria.

Observaciones.

Esta implementación final de la clase surgió fruto del trabajo sobre otros enfoques intermedios, que se considera pertinente plasmar para dar una visión global del proceso de desarrollo de este programa.

Inicialmente, esta gestión de la memoria dinámica usada por el programa se intentó hacer mediante asignación manual, tal y como se hacía en `C`, pero resultó en una gran cantidad de problemas de gestión de la memoria.

Parte del problema estaba en que inicialmente se consideró hacer un *distance field* para cada estado de carga del problema. La idea era que se necesitaba hacer esto para interpolar la distancia que tendría el punto cuando la barra estaba deformada e indeformada. Sin embargo, tal y como opera la función de cálculo de desplazamientos, se vio que no era necesario hacer esto, y que simplemente eran necesarias las distancias relativas al sólido indeformado (ya que se calculan los desplazamientos respecto al caso indeformado).

Intentar calcular un *distance field* para cada estado de carga, junto al uso de un índice de para cada dirección (i, j y k), daba una array dinámica de cuatro índices. El uso de una asignación manual de memoria dinámica daba problemas de compilación, ya que era necesario definir una serie de vectores de punteros interrelacionados que producían inconsistencia de tipos (por ejemplo, Visual Studio intentaba convertir punteros tipo `double*` a `double**`).

Antes incluso de eliminar el índice dedicado a los estados de carga, se decidió constreñir los tres índices de la matriz *distance field* en uno solo, y usar una función para mimetizar el acceso a elementos que tendría con tres índices. De esta manera reducíamos el número de capas de punteros a dos. La función en cuestión, llamada `Coor2N`, realiza la siguiente operación:

```
i + j*_Nx + k*_Nx*_Ny;
```

Siendo i, j, k los índices del elemento al que queremos acceder, y N_x, N_y son el número de elementos en x y en y. Esta función, que es la que usa internamente C/C++ cuando se define un array con más de un índice, asegura el acceso al bloque de memoria buscado.

Posteriormente se dejó la asignación manual de la memoria dinámica, que seguía dando problemas al intentar liberar la memoria, en favor de la clase `vector` proporcionada por la *standard library*. Dicha clase gestiona la memoria dinámica de la que hace uso y la libera ella misma sin que el programador tenga que preocuparse. Esta funcionalidad, conocida en informática como *colector de basura* (donde la basura es la memoria dinámica que una vez usada, ocupa espacio inútilmente en la RAM), eliminaba los problemas que daba la asignación manual. Cuando se determinó que sólo era necesario el *distance field* de un estado (el indeformado), la variable dedicada a almacenar las distancias calculadas se redujo a un simple `vector<double>`.

3 CLASE SOLID

En el caso de `Solid`, las diferencias respecto a la versión usada en la simulación son menores, principalmente se suprimen las funciones de dibujo ya que no se va a pintar la viga (al menos en principio, versiones más avanzadas del programa podrían incluir una interfaz gráfica).

Aunque pudiera parecer que no es necesario incluir esta clase, sino simplemente el fichero `malla.h`, lo cierto es que facilita la obtención de normales a las caras de la superficie (necesario para distinguir entre el volumen interno y el externo al sólido), y además, en caso de querer calcular un *distance field* respecto a algún estado de carga en concreto, es necesario tener la clase para calcular las deformaciones.

4 OBTENCIÓN DEL CAMPO DE DISTANCIAS

Conceptualmente, se trata de obtener la distancia euclídea de cada punto del *distance field* al punto más cercano del sólido. Además, hay que tener en cuenta la traslación que se da al *distance field* para ajustarlo al sólido.

Como ya se comentó en la introducción al capítulo, se debe encontrar el vértice S de la superficie del sólido que sea más cercano al punto P de la malla del *distance field*. En concreto, el algoritmo recorre todos los nodos del *distance field* en un bucle y en cada iteración i de dicho bucle se obtiene el vector que une el origen de coordenadas $\mathcal{O}$ con el punto P_i del *distance field*. Después, se obtiene el vector que une el origen con el punto de la superficie S más cercano a P_i. El vector que une P_i con S se llamará $\vec{v} = \vec{p} - \vec{s}$ y es el que contiene la distancia del P_i a la superficie, como se ve en la figura.

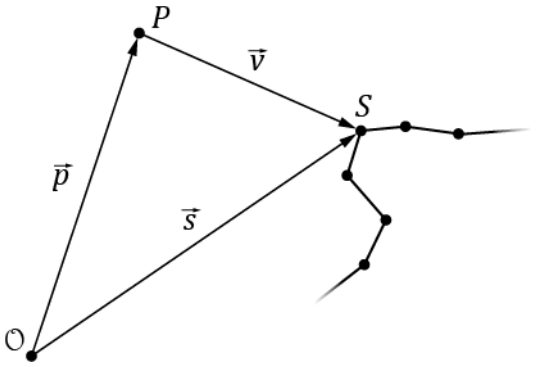


Figura 15. Vector que une P con S.

El valor que toma el *distance field* será el módulo de $\vec{v}$, $\|\vec{v}\|$.

5 DETECCIÓN DENTRO-FUERA

Inicialmente, la detección dentro-fuera se hacía usando las normales a los vértices. Pueden obtenerse a partir de las normales a las caras, haciendo la suma de las normales a todas las caras a las cuales pertenece el vértice, y normalizando la resultante. Al multiplicar la normal en el vértice por $\vec{v}$, se obtiene la distancia con signo.

Debido a incoherencias en el cálculo de las normales en puntos pertenecientes a esquinas, esta forma de trabajar detectaba algunos puntos del *distance field* como internos al sólido, cuando en realidad estaban fuera. En ciertas geometrías no hay problema, pero sólidos con ángulos entre aristas rectos (o agudos), como un prisma, resultan afectados.

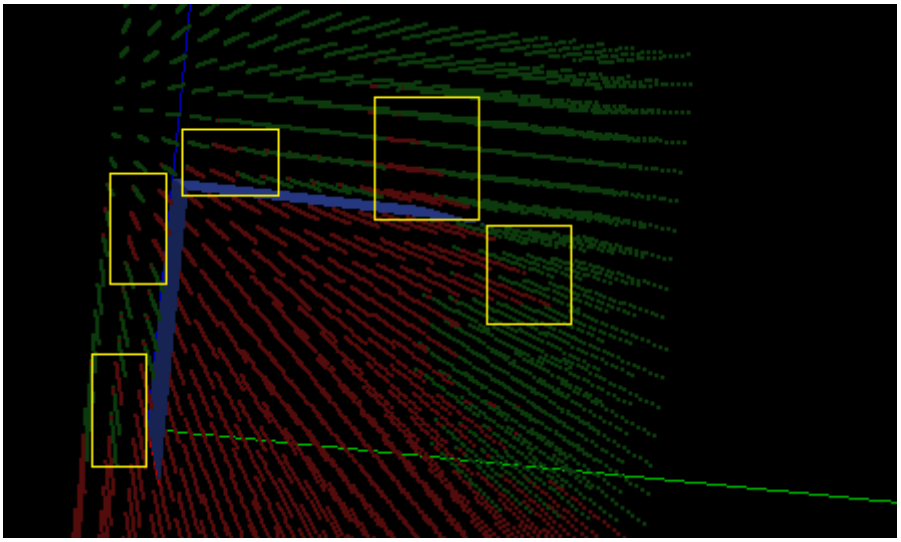
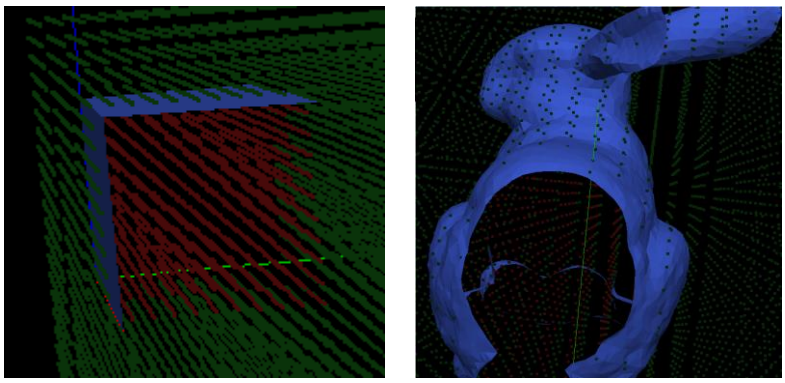


Figura 16. Detección como puntos internos al sólido de nodos que están fuera realmente.

Se decidió pasar a un enfoque similar, aunque menos refinado: simplemente, se comprueban las normales a todos los triángulos que contienen al nodo más cercano. El criterio que decide la pertenencia al interior del sólido es el siguiente: si todos los productos $\vec{n} \cdot \vec{v}$ son ≤ 0 (criterio que se sigue para determinar que P está dentro del sólido), se interpreta que está dentro. En caso de que un producto fuese positivo, P está fuera del sólido.

Con ello, se obtiene correctamente la pertenencia.



a. *b.*

Figura 17. Detección correcta en (a) Viga y (b) StandfordBunny

Así se completa el cálculo del *distance field*.

6 SALIDA

La función `WriteFile` genera un archivo de cabecera de C/C++, denominado `LSData.h`, con la siguiente información:

- Número de elementos en x, y y z.
- Espaciado entre elementos en cada dirección.
- Una matriz llamada `LSVertices` que contiene las coordenadas de cada nodo de la malla del *distance field* (con la traslación que se deba dar para ajustarlo al sólido).

- Una matriz llamada `LSMatrix` que contiene las distancias calculadas con el signo correspondiente.

La variable `LSVertices` se considera pertinente para hacer más fácil el acceso a la información geométrica sin rotar. Esto se usa principalmente para dos cosas: en la interpolación con funciones de forma y para calcular una variable, `vertices`, que contiene las coordenadas de los vértices con la rotación y traslación que corresponda.

6

RESULTADOS

En éste capítulo se presentan los resultados de la simulación. El material usado para el análisis mediante PGD era elástico lineal, bajo hipótesis de Kirchoff—Saint Venant, con un módulo elástico $E = 2 \cdot 10^6$ N/m^2 y coeficiente de Poisson $\nu = 0.3$.

La carga es igual para todos los nodos y tiene un valor de 300 N.

En la siguiente figura se representa la malla del sólido. Los puntos que aparecen en rojo están empotrados.

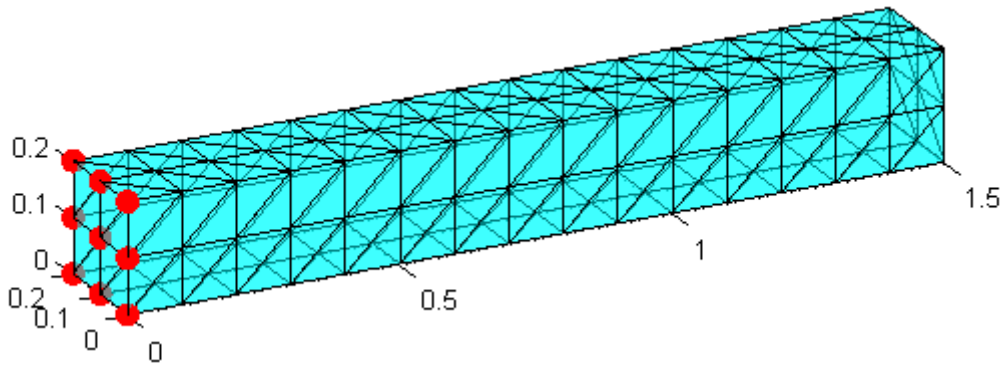


Figura 18. Malla del sólido.

Como se puede ver, las medidas del sólido son 1.5 m × 0.2 m × 0.2 m.

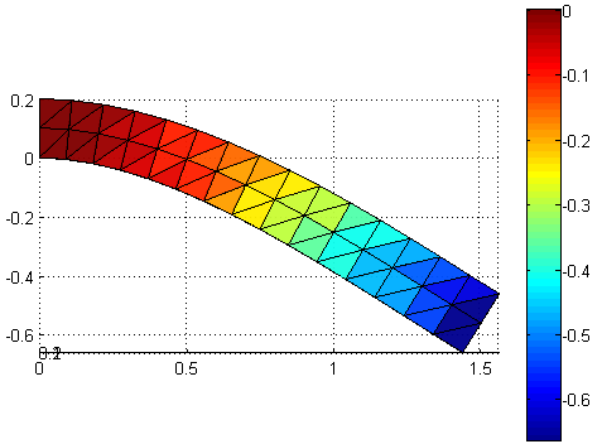


Figura 19. Deformación de la viga si la carga se coloca en un nodo del extremo.

En la siguiente se representa la configuración inicial de los dos sólidos, tal y como se vio en el Capítulo 5.

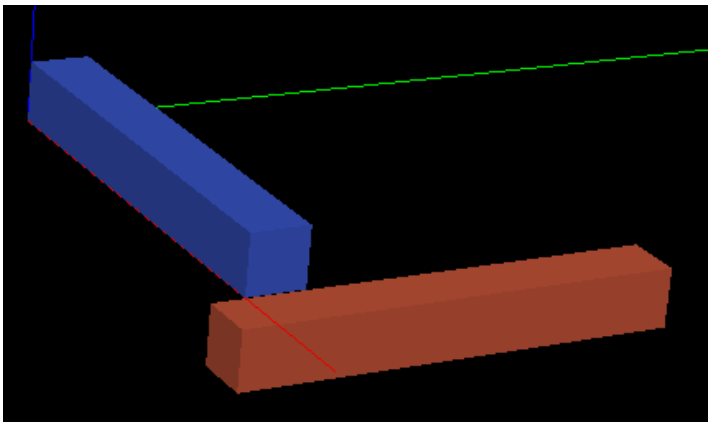


Figura 20.

Al tocar la cara superior del prisma, se aplica una deformación sobre la viga. Cuando ésta entra en conctacto con el segundo prisma también se deforma.

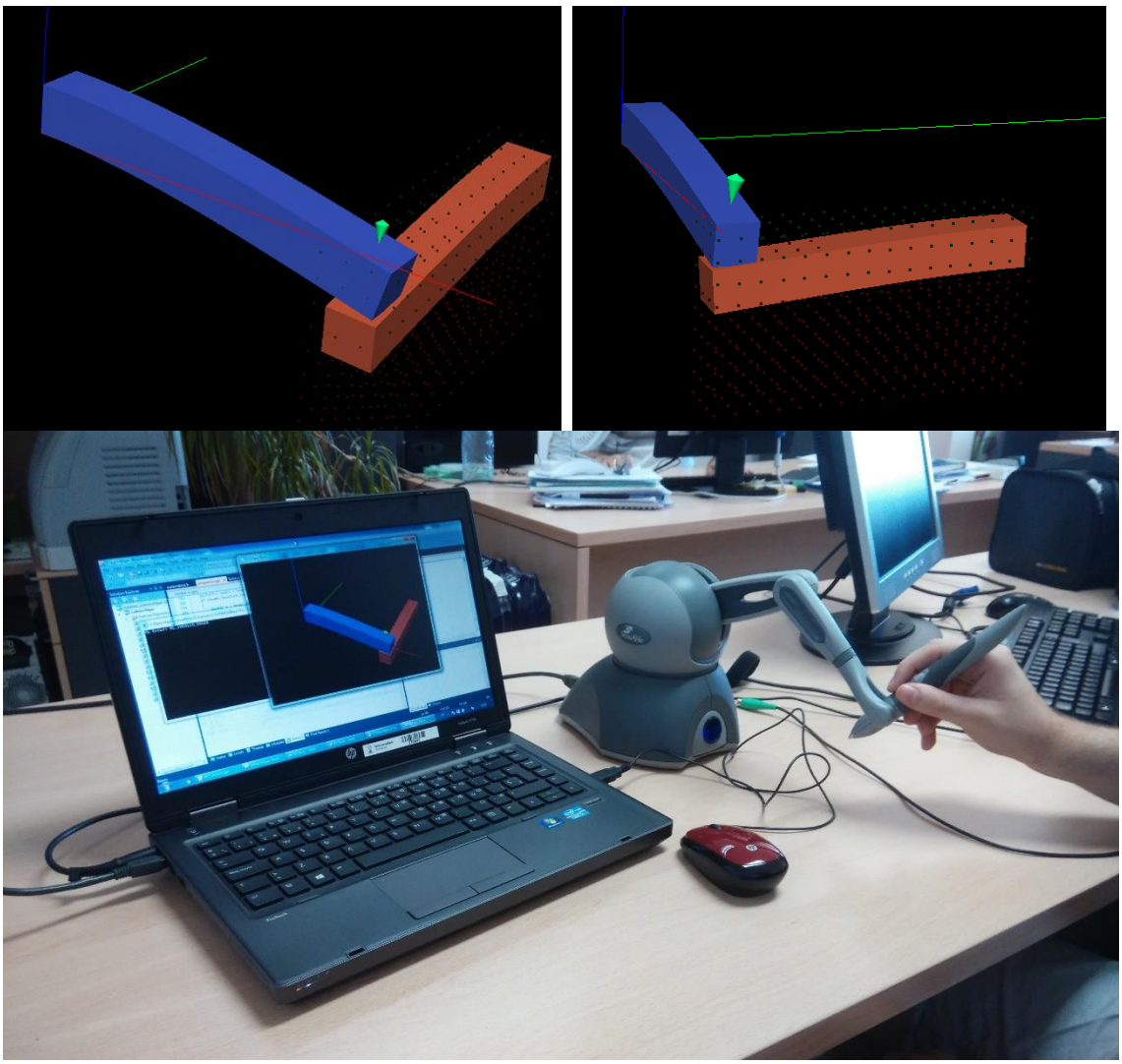


Figura 21. Contacto entre los sólidos.

7

CONCLUSIONES Y LÍNEAS FUTURAS

1 OBJETIVOS DEL PROYECTO

Como se dijo en la introducción, el propósito del TFG es probar que es posible alcanzar una frecuencia de actualización del estado de la simulación de 500 a 1000 Hz para el caso de dos sólidos deformables que pueden interactuar entre ellos.

A la luz de las simulaciones realizadas, se considera que, efectivamente, se logran dichas tasas de actualización. Aunque en este trabajo no se ha medido exactamente la velocidad de respuesta, se ha comprobado que el usuario percibe la respuesta háptica sin los saltos típicos que se dan cuando la respuesta es demasiado lenta, no percibe una respuesta discreta sino continua.

El método PGD demuestra su potencial para éste tipo de aplicaciones, y se abre la puerta a problemas de más alta dimensionalidad (por ejemplo: incluir comportamiento plástico).

2 LÍNEAS FUTURAS

En éste apartado se indican posibles áreas de mejora y sugerencias sobre donde enfocar trabajos futuros sobre el código.

2.1 Clase Vector3

Ya que la funcionalidad de `Vector3` es principalmente geométrica, podrían hacerse algunos cambios que lo optimizarían a este respecto y, haciendo uso de la herencia, obtener tanto una clase "Vector" (que efectivamente represente un vector en el sentido estricto) como una clase "Punto".

Para ello, se definirá un cuarto índice, no accesible mediante subíndice, que tendrá el valor 0 si es un vector y 1 si es un punto. Esta forma se usa mucho en gráficos por ordenador, ya que las operaciones de traslación y rotación se fusionan en una matriz 4x4 de la forma:

$$\boldsymbol{A} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

De esta forma, podemos rotar un vector $v^T = (x \quad y \quad z \quad 0)$ al multiplicarlo por $\boldsymbol{A}$, pero no será trasladado (algo lógico, ya que un vector es básicamente una dirección y una magnitud en el espacio, por lo que no tiene sentido hablar de trasladar vectores), y un punto $P^T = (x \quad y \quad z \quad 1)$ sufrirá además de la rotación, la traslación guardada en la cuarta columna de $\boldsymbol{A}$. En las siguientes ecuaciones se ponen los resultados de multiplicar $\boldsymbol{A}$ por un punto y por un vector, respectivamente.

$$\boldsymbol{A} \cdot P = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \cdot r_{11} + y \cdot r_{12} + z \cdot r_{13} + t_1 \\ x \cdot r_{21} + y \cdot r_{22} + z \cdot r_{23} + t_2 \\ x \cdot r_{31} + y \cdot r_{32} + z \cdot r_{33} + t_2 \\ 1 \end{pmatrix}$$

$$\boldsymbol{A} \cdot v = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} = \begin{pmatrix} x \cdot r_{11} + y \cdot r_{12} + z \cdot r_{13} \\ x \cdot r_{21} + y \cdot r_{22} + z \cdot r_{23} \\ x \cdot r_{31} + y \cdot r_{32} + z \cdot r_{33} \\ 1 \end{pmatrix}$$

Esta clase es usada por todas las demás, por lo que cualquier cambio en ella tendrá profundo impacto en el código.

2.2 Clase RotM

En relación a los cambios sugeridos en `Vector3`, la clase sería reformulada para incluir traslaciones, es decir, contendría un array 4×4 de forma que sus componentes fuesen:

$$\begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

2.3 Clase Solid

Como ya se dijo en el Capítulo 4, ésta clase probablemente pueda ser separada en varias. Lo que se propone es, principalmente, eliminar el código encargado de la parte gráfica y pasarlo todo bien a funciones del archivo `Programa`, bien (la opción preferida) a una clase nueva que oculte el funcionamiento de OpenGL (por ejemplo, una clase llamada `GL_Interface`) y cuyas entradas sean entidades a representar. Lo mismo puede decirse de la parte háptica del código.

Con ello, la clase Solid se convertiría en un contenedor de geometría, PGD y distance field (las tres entidades que definen un sólido en el modelo de simulación, como se expuso en el Capítulo 4), y sus funciones serían exclusivamente de cálculo de deformaciones.

Para su representación, un método `GetGeometry` devolvería los vértices y normales (ya deformados), y éstos serían los datos de entrada (junto con una traslación y rotación) a la clase `GL_Interface`, la cual haría las operaciones necesarias para el renderizado en pantalla.

2.4 Más encapsulación

Un aspecto fundamental de la encapsulación del código es que permite la especialización del programador, asi como eliminar las "distracciones" que puede provocar la mezcla de diversas funcionalidades en un mismo archivo.

Evidentemente, encapsular todas las funciones de gestión gráfica y háptica en una o dos clases separadas permitiría a los ingenieros mecánicos concentrarse en la parte de modelado del sistema físico, es decir el cálculo de las deformaciones y el comportamiento del sólido en general.

Para lograr la encapsulación, es necesario representar los flujos de información de la aplicación y agruparlos según la función que tienen en la ejecución. El siguiente diagrama es un esquema de los flujos de información más importantes.

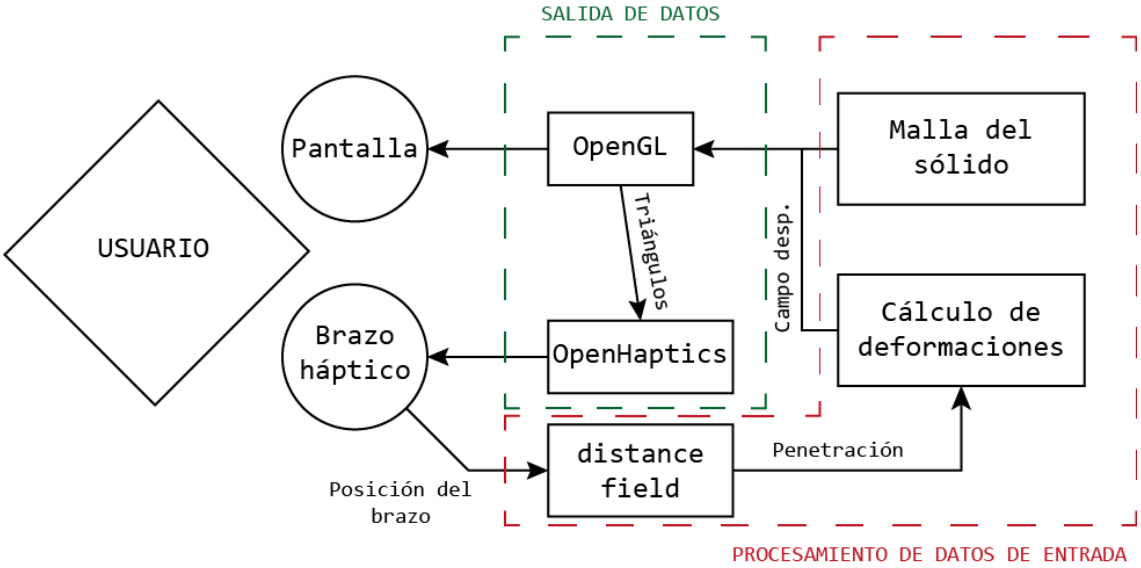


Figura 22. Flujo de la información en la aplicación.

REFERENCIAS

[1] R. Satava, «Medical Virtual Reality: the current status of the future,» 1996.

[2] W. A. McNeely, K. D. Puterbaugh y J. J. Troy, «Six Degree-of-Freedom Haptic Rendering Using Voxel Sampling.».

[3] J. Barbic y D. L. James, «Six-DoF Haptic Rendering of Contact between Geometrically Complex Reduced Deformable Models,» *IEEE TRANSACTIONS ON HAPTICS,* vol. 1, nº 1, pp. 39-52, 2008.

[4] I. A. C. Q. E. C. F. C. David González, «Computational vademecums for the real-time simulation of haptic collision between nonlinear solids.,» *Computer methods in applied mechanics and engineering,* nº 283, pp. 210-223, 2014.

[5] A. Ammar, B. Mokdad, F. Chinesta y R. Keunings, «A new family of solvers for some cases of multidimensional partial differential equations encountered in kinetic theory modeling of complex fluids,» *Journal of Non-Newtonian Fluid Mechanics,* nº 139, pp. 153-176, 2006.

[6] P. Ladeveze, «Nonlinear Computational Structural Mechanics-New Approaches and Non-Incremental Methods of Calculation,» p. 220, 1999.

[7] P. Ladeveze, J. C. Passieux y D. Neron, «The latin multiscale computational method and the proper generalized decomposition,» *Computer Methods in Applied Mechanics and Engineering,* vol. 199, nº 21-22, pp. 1287-1296, 2010.

[8] F. Chinesta, R. Keunings y A. Leygue, «The Proper Generalized Decomposition at a Glance,» de *The Proper Generalized Decomposition for Advanced Numerical Simulations: A Primer*, Springer, 2014, pp. 10-12.

[9] Department of Aerospace Engineering Sciences of the University of Colorado, «11 Hexahedron elements,» de *Advanced Finite Element Methods for Solids, Plates and Shells*, pp. 11-5.

ANEXO A

RESOLVIENDO UN PROBLEMA MEDIANTE PGD

En el presente Anexo se desarrolla en mayor profundidad la metodología de resolución de un problema mediante PGD. En concreto, se ha elegido el ejemplo propuesto en [4].

Como punto de partida, se considera el caso de un vademécum en el que interesa almacenar el campo de desplazamientos $u(\boldsymbol{x})$ de un sólido Ω bajo la acción de una fuerza (que se asume unitaria y siempre de dirección $-Z$ para simplificar el problema) en cualquier punto s de la frontera $\overline{\Gamma}$.

Esto deja el problema definido en general en $\mathbb{R}^5$ $(u = u(\boldsymbol{x}, \boldsymbol{s}))$, aunque si s se interpola por el vecino más cercano, puede verse como un parámetro unidimensional (el nodo donde actúa la carga), dejando el problema en $\mathbb{R}^4$.

Por simplicidad, y para mostrar con mayor claridad las particuaridades de la *Proper Orthogonal Decomposition*, se va a presentar la formulación del problema elástico lineal, ignorando los términos de inercia.

Bajo dichas suposiciones, la formulación débil del problema, extendida al sólido Ω y a la porción de la frontera accesible a la carga, $\overline{\Gamma} \subset \Gamma_t$, consiste en encontrar el desplazamiento $u \in \mathcal{H}^1$ tal que para todo $u^* \in \mathcal{H}_0^1$ se cumpla la siguiente ecuación:

$$\int_{\overline{\Gamma}} \int_{\Omega} \nabla_s \boldsymbol{u}^* : \boldsymbol{\sigma} \, d\Omega \, d\overline{\Gamma} = \int_{\overline{\Gamma}} \int_{\Gamma_{t2}} \boldsymbol{u}^* \cdot \boldsymbol{t} \, d\Gamma \, d\overline{\Gamma} \qquad (4)$$

donde $\Gamma = \Gamma_u \cup \Gamma_t$ representa la frontera del sólido, dividida en región esencial y natural, y donde $\Gamma_t = \Gamma_{t1} \cup \Gamma_{t2}$, es decir, regiones con condiciones de frontera homogéneas y no-homogéneas, respectivamente. Aquí el vector tensión no es un dato, como suele ser tradicionalmente en este tipo de formulacones, sino que es el vector tensión en cualquier punto s de la frontera $\overline{\Gamma}$, por lo que su expresión es $\boldsymbol{t} = -\boldsymbol{e}_k \cdot \delta(\boldsymbol{x} - \boldsymbol{s})$, donde δ representa la delta de Dirac y $\boldsymbol{e}_k$ el vector unidad en el eje Z.

La delta de Dirac es regularizada y normalizada mediante una serie truncada de funciones separables, según la filosofía del método PGD.

$$t_j \approx \sum_{i=1}^{m} f_j^i(\boldsymbol{x}) g_j^i(s)$$

donde m representa el orden del truncado y f_j^i, g_j^i representan la *j*-ésima componente de funciones vectoriales en el espacio y en la posición en la frontera, respectivamente.

Las técnicas PGD permiten construir de forma eficiente el vademécum computacional de $u(\boldsymbol{x}, s)$ construyendo, de forma iterativa, una aproximación a la función solución en la forma de una suma finita de funciones separables.

Supongamos que mediante pgd se ha llegado a la siguiente aproximación en la iteración n:

$$u_j^n(\boldsymbol{x}, s) = \sum_{k=1}^{n} X_j^k(\boldsymbol{x}) \cdot Y_j^k(s)$$

donde el término u_j se refiere a la j-ésima componente del vector de desplazamientos, $j = 1,2,3$ y las funciones $X_j^k(\boldsymbol{x})$ y $Y_j^k(s)$ representan las funciones separadas usadas para aproximar el campo desconocido, obtenidas en iteraciones previas del algoritmo de PGD. El objetivo de PGD es proveer una solución mejorada dada por el término $n + 1$ de la aproximación.

$$u_j^{n+1}(\boldsymbol{x}, s) = u_j^n(\boldsymbol{x}, s) + R_j(\boldsymbol{x}) \cdot S_j(s)$$

donde $R_j(\boldsymbol{x})$ y $S_j(s)$ son funciones que mejoran la aproximación, incógnitas del problema para esta iteración.

De forma análoga, las variaciones admisibles en el campo de desplazamientos vendrán dadas por:

$$u_j^*(\boldsymbol{x}, s) = R_j^*(\boldsymbol{x}) \cdot S_j(s) + R_j(\boldsymbol{x}) \cdot S_j^*(s)$$

Introduciendo las separaciones de la carga, del desplazamiento y de su variación admisible en la formulación débil del problema, y teniendo la relación lineal $\boldsymbol{\sigma} = \boldsymbol{C}: \nabla_s u$, donde $\boldsymbol{C}$ es el tensor de comportamiento, se obtiene la siguiente expresión:

$$\int_{\overline{\Gamma}} \int_{\Omega} \nabla_s \left(R_j^*(\boldsymbol{x}) \cdot S_j(s) + R_j(\boldsymbol{x}) \cdot S_j^*(s) \right) : \boldsymbol{C} : \nabla_s \left(u_j^n(\boldsymbol{x}, s) + R_j(\boldsymbol{x}) \cdot S_j(s) \right) \, d\Omega \, d\overline{\Gamma}$$
$$= \int_{\overline{\Gamma}} \int_{\Gamma_{t2}} \left(R_j^*(\boldsymbol{x}) \cdot S_j(s) + R_j(\boldsymbol{x}) \cdot S_j^*(s) \right) \cdot \left(\sum_{i=1}^{m} f_j^i(\boldsymbol{x}) g_j^i(s) \right) d\Gamma \, d\overline{\Gamma}$$

Como se puede apreciar, incluso aunque el problema de partida es lineal, PGD necesita solucionar un problema no lineal, es decir, determinar un producto de funciones $R_j(\boldsymbol{x}) \cdot S_j(s)$. A éste efecto, podría usarse cualquier metodología de linealización, pero se elige el método de punto fijo por su sencillez y velocidad.

El algoritmo de punto fijo procede de forma iterativa enriqueciendo la solución. En cada paso n del enriquecimiento (para $n \geq 1$), los primeros $n - 1$ términos de la aproximación PGD descrita ya se suponen calculados:

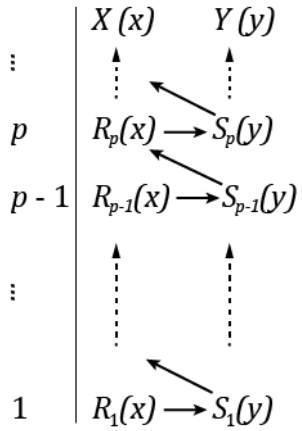
$$u^{n-1}(\boldsymbol{x}, s) = \sum_{j=1}^{n-1} X_j(\boldsymbol{x}) \cdot Y_j(s)$$

A continuación, se desea calcular el siguiente término de la aproximación:

$$u^n(\boldsymbol{x}, s) = u^{n-1}(\boldsymbol{x}, s) + R(\boldsymbol{x}) \cdot S(s) = \sum_{i=1}^{n-1} X_j(\boldsymbol{x}) \cdot Y_j(s) + R(\boldsymbol{x}) \cdot S(s)$$

Tanto $R(\boldsymbol{x})$ como $S(s)$ son funciones desconocidas en el paso actual n del enriquecimiento. Puesto que aparecen en forma de producto, el problema resultante es no lineal y requiere de un esquema de linealización adecuado.

El algoritmo de punto fijo permite obtener la $R(x)$ de la iteración actual p a partir de la $S(y)$ de la iteración anterior $p-1$ para, a continuación, obtener la $S(y)$ de la iteración actual a partir de la $R(x)$ que se acaba de calcular.



Para que comience el proceso iterativo, se debe especificar un valor inicial arbitrario $S_0(y)$. Las iteraciones se sucederán hasta alcanzar un punto fijo, delimitado por una tolerancia ϵ previamente especificada, es decir:

$$\frac{\left\| R_p(x) \cdot S_p(y) - R_{p-1}(x) \cdot S_{p-1}(y) \right\|}{R_{p-1}(x) \cdot S_{p-1}(y)} < \epsilon$$

Donde $\|\cdot\|$ es una norma adecuada.

Una vez halladas las funciones, el paso n del enriquecimiento termina identificando $R(x)$ con $X_n(x)$ y $S(y)$ con $Y_n(y)$.

El proceso de enriquecimiento se detiene cuando se obtiene una determinada medida del error $e(n)$ lo bastante pequeña, es decir $e(n) < \tilde{\epsilon}$.

2.1 Cálculo de $S(s)$ asumiendo que $R(\boldsymbol{x})$ es conocida.

En este caso, tenemos:

$$u_j^*(\boldsymbol{x}, s) = R_j(\boldsymbol{x}) \cdot S_j^*(s)$$

o, equivalentemente, $\boldsymbol{u}^*(\boldsymbol{x}, s) = \boldsymbol{R} \circ \boldsymbol{S}^*$, donde el símbolo "$\circ$" es el producto de Hadamard, o producto término a término de matrices. Una vez sustituido en la Ec. (4), da

$$\int_{\overline{\Gamma}} \int_{\Omega} \nabla_s(\boldsymbol{R} \circ \boldsymbol{S}^*) : \boldsymbol{C} : \nabla_s \left(\sum_{k=1}^{n} \boldsymbol{X^k} \cdot \boldsymbol{Y^k} + \boldsymbol{R} \circ \boldsymbol{S} \right) d\Omega \, d\overline{\Gamma} =$$

$$= \int_{\overline{\Gamma}} \int_{\Gamma_{t2}} (\boldsymbol{R} \circ \boldsymbol{S}^*) \cdot \left(\sum_{k=1}^{n} \boldsymbol{f} \cdot \boldsymbol{g} \right) d\Gamma \, d\overline{\Gamma}$$

o, equivalentemente:

$$\int_{\overline{\Gamma}} \int_{\Omega} \nabla_s(\boldsymbol{R} \circ \boldsymbol{S}^*) : \boldsymbol{C} : \nabla_s(\boldsymbol{R} \circ \boldsymbol{S})\, d\Omega\, d\overline{\Gamma} =$$
$$= \int_{\overline{\Gamma}} \int_{\Gamma_{t2}} (\boldsymbol{R} \circ \boldsymbol{S}^*) \cdot \left(\sum_{k=1}^{n} \boldsymbol{f} \cdot \boldsymbol{g} \right) d\Gamma\, d\overline{\Gamma} - \int_{\overline{\Gamma}} \int_{\Omega} \nabla_s(\boldsymbol{R} \circ \boldsymbol{S}^*) \cdot \mathcal{R}^n\, d\Omega\, d\overline{\Gamma}$$

donde $\mathcal{R}^n = \boldsymbol{C} : \nabla_s \boldsymbol{u}^n$.

Como el gradiente simétrico sólo opera en variables espaciales, se tiene:

$$\int_{\overline{\Gamma}} \int_{\Omega} (\nabla_s \boldsymbol{R} \circ \boldsymbol{S}^*) : \boldsymbol{C} : (\nabla_s \boldsymbol{R} \circ \boldsymbol{S})\, d\Omega\, d\overline{\Gamma} =$$
$$= \int_{\overline{\Gamma}} \int_{\Gamma_{t2}} (\boldsymbol{R} \circ \boldsymbol{S}^*) \cdot \left(\sum_{k=1}^{n} \boldsymbol{f} \cdot \boldsymbol{g} \right) d\Gamma\, d\overline{\Gamma} - \int_{\overline{\Gamma}} \int_{\Omega} (\nabla_s \boldsymbol{R} \circ \boldsymbol{S}^*) \cdot \mathcal{R}^n\, d\Omega\, d\overline{\Gamma}$$

Todos los términos dependientes de $\boldsymbol{x}$ son conocidos por lo que es posible obtener el valor de s que cumple la ecuación para cualquier variación admisible S^*.

Generalmente se utiliza una aproximación de elementos finitos en las variables R y S para resolver la ecuación de manera aproximada.

2.2 Cálculo de $R(\boldsymbol{x})$ asumiendo que $S(s)$ es conocida.

Equivalentemente, en éste caso se tiene:

$$u_j^*(\boldsymbol{x}, s) = R_j^*(\boldsymbol{x}) \cdot S_j(s)$$

Que, una vez más, al ser sustituida en la Ec. (4) da:

$$\int_{\overline{\Gamma}} \int_{\Omega} \nabla_s(\boldsymbol{R}^* \circ \boldsymbol{S}) : \boldsymbol{C} : \nabla_s \left(\sum_{k=1}^{n} \boldsymbol{X^k} \cdot \boldsymbol{Y^k} + \boldsymbol{R} \circ \boldsymbol{S} \right) d\Omega\, d\overline{\Gamma} =$$
$$= \int_{\overline{\Gamma}} \int_{\Gamma_{t2}} (\boldsymbol{R}^* \circ \boldsymbol{S}) \cdot \left(\sum_{k=1}^{n} \boldsymbol{f} \cdot \boldsymbol{g} \right) d\Gamma\, d\overline{\Gamma}$$

En este caso todos los términos dependientes de $\boldsymbol{s}$ (posición de la carga) pueden ser integrados sobre $\overline{\Gamma}$, lo que lleva a un problema elástico generalizado para calcular $R(\boldsymbol{x})$ que se resuelve de manera similar al caso anterior.

Este desarrollo asume pequeñas deformaciones. Para las ecuaciones generales de hiperelasticidad, los tensores de grandes deformaciones (usualmente el tensor de Green-Lagrange $\boldsymbol{E}$) debe ser igualmente linealizado.

ANEXO B

CÓDIGO DEL SIMULADOR

1 ARCHIVO Programa.cpp

```
/*****************************************************************************

Copyright (c) 2004 SensAble Technologies, Inc. All rights reserved.

OpenHaptics(TM) toolkit. The material embodied in this software and use of
this software is subject to the terms and conditions of the clickthrough
Development License Agreement.

For questions, comments or bug reports, go to forums at:
    http://dsc.sensable.com

*****************************************************************************/

#include <stdlib.h>
#include <math.h>
#include <assert.h>
#include <vector>
#include <iostream>


#if defined(WIN32)
#include <windows.h>
#endif

#if defined(WIN32) || defined(linux)
#include <GL/glut.h>
#elif defined(__APPLE__)
#include <GLUT/glut.h>
#endif


//data files
#include "Vector3.h"
#include "Solid.h"
#include "mysettings.h"

#include <HL/hl.h>
#include <HD/hd.h>
#include <HDU/hduMath.h>
#include <HDU/hduMatrix.h>
#include <HDU/hduQuaternion.h>
```

```
#include <HDU/hduError.h>
#include <HLU/hlu.h>
#include <HDU/hduVector.h>


#define FORCE 1000


//NOTA: LAS FUNCIONES CON ********************************************** SON
LAS QUE HAY QUE TOCAR EN PGD




/* Haptic device and rendering context handles. */
static HHD ghHD = HD_INVALID_HANDLE;
static HHLRC ghHLRC = 0;

/* Shape id for shape we will render haptically. */
HLuint gHapticShapeId;

/* desplazamientod de todos los nodos de la superficie */
//GLdouble displSup[3][NumNodesSup] = {0.0};
/* desplazamiento de la superficie cargada. */
//GLdouble displS[3][NumNodesS] = {0.0};


/* Cursor constants */
#define CURSOR_SIZE_PIXELS 20
static double gCursorScale;
static GLuint gCursorDisplayList = 0;


//listado de funciones, por orden de aparicion
/* Function prototypes. Se las llama en main*/
void glutDisplay(void);//manda pintar drawSceneHaptics y drawSceneGraphics
void glutReshape(int width, int height);//ajusta la vista cuando cambiamos el ta-
maño de la ventana
void      glutIdle(void);//Brazo      haptico.      Llama      a      GestionaToque
**************************************************
void glutMouse(int button, int state, int x, int y);//detecta si hay boton de raton
pulsado y activa las tareas de zoom, traslacion o rotacion
void glutMotion(int x, int y);//realiza zoom, traslacion y rotacion

//resto de funciones
double projectToTrackball(double radius, double x, double y);//usada en glutMotion
para la rotacion
void updateCamera();//actualiza la posicion de la camara
void updateHapticMapping();//??? usada en updateCamera
void exitHandler(void);//cierra, limpia y sale del programa

void initScene();//llama a initGL e initHL
void initGL();//inicializa propiedades de la escena grafica (luces, ...)
void initHL();//inicializa propiedades de la escena haptica (fuerzas, ...)
void drawSceneGraphics();//pinta la escena grafica: viga deformada y cursor
void drawSceneHaptics();//"pinta" la escena haptica y devuelve la fuerza al brazo
haptico **********************************************
//void drawObject();//pinta la viga deformada. El calculo del campo de desplaza-
mientos ya se ha hecho en getObjectDisplacement
//void drawLoadedSurface();//pinta la superficie de contacto en configuracion de-
formada
```

```
void drawCursor();//pinta un cono para el cursor en la posicion del brazo haptico
void DrawAxis(); //Pinta los ejes de coordenadas.


//gestiona el contacto entre el brazo y el objeto. Comprueba si se toca y en que
nodo. Calcula el
//desplazamiento de ese nodo y de todos los nodos de la viga. Llama a getClosestNode
y a getObjectDisplacement
void GestionaToque();// ************************************************
void GestionaColision();
//funcion auxiliar que gestiona la sensacion haptica cuando se ha dejado de tocar
la viga
void Descarga(int);// ************************************************
void DescargaOK(int);
//busca el nodo más cercano al proxi y la posicion del proxi. -1 significa que ha
habido algun error
//void getClosestNode();// ************************************************
//para un determinado desplazamiento de un nodo cargado, calcula el desplazamiento
de todos los nodos de la viga
//void                                              getObjectDisplacement();//
**********************************************


//Variables globales
HLboolean isTouching = false;//obtenida en GestionaToque y usada en GestionaToque
y drawSceneHaptics
HLboolean is1stTimeHere = true;//usada en drawSceneHaptics. Para inicializar la
fuerza solo la primera vez

HLboolean firstContact = true;
GLdouble InitialProxyPosition[3];//posicion del brazo haptico. Calculada en Ges-
tionaToque
GLdouble proxyPos[3];
GLdouble displacement=0;//desplazamiento del nudo cargado en dirección -Z  (posi-
tivo si va en -z)
GLdouble LoadScale;//displacement and load scale factor
//int loadedNodeS = -1;//nodo, de la superficie cargada S, donde se aplica la
carga. La numeracion empieza en 0. Calculada en getClosestNode



/* Effect ID */
HLuint gEffect;


static hduVector3Dd gCameraPosWC;
static int gWindowWidth, gWindowHeight;

/* Variables used by the trackball emulation. */
//Originalmente la vista era desde el eje z positivo, plano xy. El eje x quedaba
hacia la derecha, el y hacia arriba, y el 0,0,0 en el centro de la pantalla
/*static hduMatrix gCameraRotation;//se inicializa por defecto como matriz identi-
dad
static double gCameraTranslationX = 0;
static double gCameraTranslationY = 0;
static double gCameraScale = 1.0;*/

//Modifico esta vista para que quede algo inclinada. Para obtener esos valores he
descomentado los valores iniciales y los comandos de imprimir que hay al principio
de la funcion upateCamera. Comienzo con la vista desde el eje z y voy girando y
trasladando hasta que tengo una vista que me gusta. Cada vez que modifico algo se
```

```
escriben en pantalla las matrices de rotación, la traslación y la escala. Cuando
tengo la vista deseada, copio los valores de pantalla aqui debajo.
static  hduMatrix  gCameraRotation(0.818974,  -0.378723,  0.431104,  0,  0.56948,
0.444069, -0.691734, 0, 0.0705358, 0.812017, 0.579356, 0, 0, 0, 0, 1);
static double gCameraTranslationX =-0.771429;
static double gCameraTranslationY = 0.114286;
static double gCameraScale = 1.1318;

//vista desde y negativo, plano xz
/*static hduMatrix gCameraRotation(1,0,0,0,0,0,-1,0,0,1,0,0,0,0,0,1);
static double gCameraTranslationX = 0;
static double gCameraTranslationY = 0;
static double gCameraScale = 1.0;*/


static bool gIsRotatingCamera = false;
static bool gIsScalingCamera = false;
static bool gIsTranslatingCamera = false;
static int gLastMouseX, gLastMouseY;


//Solid beam1(0.0, 0.0, 0.0, 0.0*kPI, 0.0*kPI, 0.0*kPI);
//Solid beam2(1.3, 1.5, -0.3, 0.0*kPI, 0.0*kPI, 1.5*kPI);
//
//#define USING_API 0
//#define COLLISION_ENABLED 0

/******************************************************************************
 Initializes GLUT for displaying a simple haptic scene.
******************************************************************************/
int main(int argc, char *argv[])
{
    // Inicializa la biblioteca GLUT
    glutInit(&argc, argv);

    // Configura el modo de visualizacin inicial
       // ventana con doble buffer, GFBA y buffer depth
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);

       // Tamagno inicial de la ventana
    glutInitWindowSize(700, 500);
       // Crea la ventana
    glutCreateWindow("PGD App Beam");


    // Set glut callback functions.
       // Callback de visualizacin para la ventana actual
    glutDisplayFunc(glutDisplay);
       // Callback de reshape para esta ventana
    glutReshapeFunc(glutReshape);
       // Callback idle
    glutIdleFunc(glutIdle);

    glutMouseFunc(glutMouse);
    glutMotionFunc(glutMotion);

    // Provide a cleanup routine for handling application exit.
    atexit(exitHandler);

    initScene();

    glutMainLoop();
```

```
    return 0;
}

/******************************************************************************
 GLUT callback for redrawing the view.
******************************************************************************/
void glutDisplay()
{
    drawSceneHaptics();

    drawSceneGraphics();

    glutSwapBuffers();
}

/******************************************************************************
 GLUT callback for reshaping the window.  This is the main place where the
 viewing and workspace transforms get initialized.
******************************************************************************/
void glutReshape(int width, int height)
{
    static const double kFovY = 20;
    static const double kCanonicalSphereRadius = 1;

    glViewport(0, 0, width, height);
    gWindowWidth = width;
    gWindowHeight = height;

    // Compute the viewing parameters based on a fixed fov and viewing
    // sphere enclosing a canonical box centered at the origin.

    double nearDist = kCanonicalSphereRadius / tan((kFovY / 2.0) * kPI / 180.0);
    double farDist = nearDist + 2.0 * kCanonicalSphereRadius;
    double aspect = (double) width / height;

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(kFovY, aspect, nearDist, farDist);

    // Place the camera.
    //la vista inicial es desde z positivo (x hacia la derecha, y hacia arriba de
la pantalla del ordenador
    gCameraPosWC[0] = 0;
    gCameraPosWC[1] = 0;
    gCameraPosWC[2] = nearDist + kCanonicalSphereRadius;

    updateCamera();
}

/******************************************************************************
 GLUT callback for idle state.  Use this as an opportunity to request a redraw.
 Checks for HLAPI errors that have occurred since the last idle check.
******************************************************************************/
void glutIdle()
{
    // Haptic error.
    HLerror error;
    while (HL_ERROR(error = hlGetError()))
    {
        fprintf(stderr, "HL Error: %s\n", error.errorCode);
```

```
        if (error.errorCode == HL_DEVICE_ERROR)
        {
            hduPrintError(stderr, &error.errorInfo,
                "Error during haptic rendering\n");
        }
    }

       GestionaToque();

#if COLLISION_ENABLED == 1
       GestionaColision();
#endif

       //if(isTouching) std::cout << loadedNodeS << " isTouching" << std::endl;
else std::cout << loadedNodeS << std::endl;

    glutPostRedisplay();
}

/*******************************************************************************

 GLUT callback for responding to mouse button presses.  Detect whether to
 initiate a point snapping, view rotation or view scale.
*******************************************************************************/
void glutMouse(int button, int state, int x, int y)
{
    if (state == GLUT_DOWN)
    {
        if (button == GLUT_LEFT_BUTTON)
        {
            gIsRotatingCamera = true;
        }
        else if (button == GLUT_RIGHT_BUTTON)
        {
            gIsScalingCamera = true;
        }
        else if (button == GLUT_MIDDLE_BUTTON)
        {
            gIsTranslatingCamera = true;
        }

        gLastMouseX = x;
        gLastMouseY = y;
    }
    else
    {
        gIsRotatingCamera = false;
        gIsScalingCamera = false;
        gIsTranslatingCamera = false;
    }
}

/*******************************************************************************
 GLUT callback for mouse motion, which is used for controlling the view
 rotation and scaling.
*******************************************************************************/
void glutMotion(int x, int y)
{
    if (gIsRotatingCamera)
    {
        static const double kTrackBallRadius = 0.8;
```

```
            hduVector3Dd lastPos;
            lastPos[0] = gLastMouseX * 2.0 / gWindowWidth - 1.0;
            lastPos[1] = (gWindowHeight - gLastMouseY) * 2.0 / gWindowHeight - 1.0;
            lastPos[2] = projectToTrackball(kTrackBallRadius, lastPos[0], lastPos[1]);

            hduVector3Dd currPos;
            currPos[0] = x * 2.0 / gWindowWidth - 1.0;
            currPos[1] = (gWindowHeight - y) * 2.0 / gWindowHeight - 1.0;
            currPos[2] = projectToTrackball(kTrackBallRadius, currPos[0], currPos[1]);

            currPos.normalize();
            lastPos.normalize();

            hduVector3Dd rotateVec = lastPos.crossProduct(currPos);

            double rotateAngle = asin(rotateVec.magnitude());
            if (!hduIsEqual(rotateAngle, 0.0, DBL_EPSILON))
            {
                hduMatrix deltaRotation = hduMatrix::createRotation(
                    rotateVec, rotateAngle);
                gCameraRotation.multRight(deltaRotation);

                updateCamera();
            }
        }
        if (gIsTranslatingCamera)
        {
            gCameraTranslationX += 10 * double(x - gLastMouseX)/gWindowWidth;
            gCameraTranslationY -= 10 * double(y - gLastMouseY)/gWindowWidth;

            updateCamera();
        }
        else if (gIsScalingCamera)
        {
            float y1 = gWindowHeight - gLastMouseY;
            float y2 = gWindowHeight - y;

            gCameraScale *= 1 + (y1 - y2) / gWindowHeight;

            updateCamera();
        }

        gLastMouseX = x;
        gLastMouseY = y;
    }

    /*******************************************************************************
     This routine is used by the view rotation code for simulating a virtual
     trackball.  This math computes the z height for a 2D projection onto the
     surface of a 2.5D sphere.  When the input point is near the center of the

     sphere, this routine computes the actual sphere intersection in Z.  When
     the input point moves towards the outside of the sphere, this routine will
     solve for a hyperbolic projection, so that it still yields a meaningful answer.

    *******************************************************************************/
    double projectToTrackball(double radius, double x, double y)
    {
        static const double kUnitSphereRadius2D = sqrt(2.0);
        double z;

        double dist = sqrt(x * x + y * y);
```

```
    if (dist < radius * kUnitSphereRadius2D / 2.0)
    {
        // Solve for sphere case.
        z = sqrt(radius * radius - dist * dist);
    }
    else
    {
        // Solve for hyperbolic sheet case.
        double t = radius / kUnitSphereRadius2D;
        z = t * t / dist;
    }

    return z;
}

/******************************************************************************
 Use the current OpenGL viewing transforms to initialize a transform for the

 haptic device workspace so that it's properly mapped to world coordinates.
******************************************************************************/
void updateCamera()
{
/*
//descomentar estas lineas si se quieren imprimir los datos de rotacion, traslacion
y escala para ajustar una nueva vista
std::cout << "gCameraRotation" << std::endl;
std::cout << gCameraRotation[0][0] << ", " << gCameraRotation[0][1] << ", " <<
gCameraRotation[0][2] << ", " <<  gCameraRotation[0][3] << std::endl;
std::cout << gCameraRotation[1][0] << ", " << gCameraRotation[1][1] << ", " <<
gCameraRotation[1][2] << ", " <<  gCameraRotation[1][3] << std::endl;
std::cout << gCameraRotation[2][0] << ", " << gCameraRotation[2][1] << ", " <<
gCameraRotation[2][2] << ", " <<  gCameraRotation[2][3] << std::endl;
std::cout << gCameraRotation[3][0] << ", " << gCameraRotation[3][1] << ", " <<
gCameraRotation[3][2] << ", " <<  gCameraRotation[3][3] << std::endl;

std::cout << "gCameraTranslation" << std::endl;
std::cout << gCameraTranslationX << ", " << gCameraTranslationY << std::endl;

std::cout << "gCameraScale" << std::endl;
std::cout << gCameraScale << std::endl;
*/

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(gCameraPosWC[0], gCameraPosWC[1], gCameraPosWC[2],
              0, 0, 0,
              0, 1,0);
    glTranslatef(gCameraTranslationX, gCameraTranslationY, 0);
    glMultMatrixd(gCameraRotation);
    glScaled(gCameraScale, gCameraScale, gCameraScale);

    updateHapticMapping();

    glutPostRedisplay();
}
/******************************************************************************
 Use the current OpenGL viewing transforms to initialize a transform for the
 haptic device workspace so that it's properly mapped to world coordinates.
******************************************************************************/
void updateHapticMapping(void)
{
    GLdouble modelview[16];
```

```
    GLdouble projection[16];
    GLint viewport[4];

    glGetDoublev(GL_MODELVIEW_MATRIX, modelview);
    glGetDoublev(GL_PROJECTION_MATRIX, projection);
    glGetIntegerv(GL_VIEWPORT, viewport);

    hlMatrixMode(HL_TOUCHWORKSPACE);
    hlLoadIdentity();

    // Fit haptic workspace to view volume.
    hluFitWorkspace(projection);

    // Compute cursor scale.
    gCursorScale = hluScreenToModelScale(modelview, projection, viewport);
    gCursorScale *= CURSOR_SIZE_PIXELS;
}

/*******************************************************************************
 This handler is called when the application is exiting.  Deallocates any state

 and cleans up.
*******************************************************************************/
void exitHandler()
{
    // Deallocate the sphere shape id we reserved in initHL.
    hlDeleteShapes(gHapticShapeId, 1);

       // Deallocate the effect id we reserved in initHL.
       hlDeleteEffects(gEffect, 1);

    // Free up the haptic rendering context.
    hlMakeCurrent(NULL);
    if (ghHLRC != NULL)
    {
        hlDeleteContext(ghHLRC);
    }

    // Free up the haptic device.
    if (ghHD != HD_INVALID_HANDLE)
    {
        hdDisableDevice(ghHD);
       }

}

/*******************************************************************************
 Initializes the scene.  Handles initializing both OpenGL and HL.
*******************************************************************************/
void initScene()
{
    initGL();
    initHL();
}

/*******************************************************************************
 Sets up general OpenGL rendering properties: lights, depth buffering, etc.
*******************************************************************************/
void initGL()
{
    static const GLfloat light_model_ambient[] = {0.3f, 0.3f, 0.3f, 1.0f};
    static const GLfloat light0_diffuse[] = {0.9f, 0.9f, 0.9f, 0.9f};
```

```
    static const GLfloat light0_direction[] = {0.0f, 0.4f, 1.0f, 0.0f};

    // Enable depth buffering for hidden surface removal.
    glDepthFunc(GL_LEQUAL);
    glEnable(GL_DEPTH_TEST);

    // Cull back faces.
    glCullFace(GL_BACK);
    glEnable(GL_CULL_FACE);

    // Setup other misc features.
    glEnable(GL_LIGHTING);
    glEnable(GL_NORMALIZE);
    glShadeModel(GL_SMOOTH);

    // Setup lighting model.
    glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_FALSE);
    glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_FALSE);
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, light_model_ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light0_diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, light0_direction);
    glEnable(GL_LIGHT0);

}

/*******************************************************************************
 Initialize the HDAPI.  This involves initing a device configuration, enabling
 forces, and scheduling a haptic thread callback for servicing the device.
*******************************************************************************/
void initHL()
{
    HDErrorInfo error;

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

    ghHLRC = hlCreateContext(ghHD);
    hlMakeCurrent(ghHLRC);

    // Enable optimization of the viewing parameters when rendering
    // geometry for OpenHaptics.

    hlEnable(HL_HAPTIC_CAMERA_VIEW);

    // Generate id for the shape.
    gHapticShapeId = hlGenShapes(1);

    hlTouchableFace(HL_FRONT);

       // Effects
       gEffect = hlGenEffects(1);
}

/*******************************************************************************

 The main routine for displaying the scene.  Gets the latest snapshot of state
```

```
 from the haptic thread and uses it to display a 3D cursor.
*****************************************************************************/
void drawSceneGraphics()

{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glClearColor(0,0,0,0);

    // Draw 3D cursor at haptic device position.
    drawCursor();

       // Draw the Axis
       DrawAxis();

    // Dibujo de la barra
    beam1.DrawGL(0.2, 0.3, 0.7, 1.0f);
       beam1.DrawLS();

       // Dibujo de la barra 2
       beam2.DrawGL(0.7, 0.3, 0.2, 1.0f);
       //beam2.DrawLS();
}

/*****************************************************************************
 The main routine for rendering scene haptics.

*****************************************************************************/
void drawSceneHaptics()
{
    // Start haptic frame.  (Must do this before rendering any haptic shapes.)
    hlBeginFrame();

       if (is1stTimeHere)
       {
              hlEffectd(HL_EFFECT_PROPERTY_MAGNITUDE, 0);
              hlStartEffect(HL_EFFECT_CONSTANT, gEffect);
              is1stTimeHere = false;
       }
       else if (isTouching)
       {
       #if HAPTIC_ENABLED
       //la direccion de la fuerza que se devuelve al brazo haptico debe ser en el
eje +z, ya que el desplazamiento es en -z
              hduVector3Dd initialDirection(0,0,1);
       //esta fuerza hay que rotarla, multiplicandola por la transpuesta de la
matriz gCameraRotation
              hduVector3Dd finalDirection;
              hduMatrix matAux = gCameraRotation.getTranspose();
              matAux.multMatrixVec(initialDirection,finalDirection);
              hlEffectdv(HL_EFFECT_PROPERTY_DIRECTION, finalDirection);

       //El valor de la fuerza que se devuelve al brazo haptico. Debe estar escalado
entre 0 (0N) y 1 (300N), por eso divido entre 300.
              hlEffectd(HL_EFFECT_PROPERTY_MAGNITUDE,   FORCE*(beam1.GetLoad()   +
beam2.GetLoad() / 2)/3000); //Va mal pero va.
              std::cout << beam1.GetLoad() << std::endl;

              hlUpdateEffect(gEffect);
       #endif
       }
       else
       {
```

```
            hlEffectd(HL_EFFECT_PROPERTY_MAGNITUDE, 0);
            hlUpdateEffect(gEffect);
        }

    // Set material properties for the shapes to be drawn.
    hlMaterialf(HL_FRONT_AND_BACK, HL_STIFFNESS, 0.0f);
    hlMaterialf(HL_FRONT_AND_BACK, HL_DAMPING, 0.0f);
    hlMaterialf(HL_FRONT_AND_BACK, HL_STATIC_FRICTION, 0.0f);
    hlMaterialf(HL_FRONT_AND_BACK, HL_DYNAMIC_FRICTION, 0.0f);
        hlMaterialf(HL_FRONT_AND_BACK, HL_POPTHROUGH, 0.0f);

        hlHintb(HL_SHAPE_DYNAMIC_SURFACE_CHANGE, HL_TRUE);

    // Start a new haptic shape.  Use the feedback buffer to capture OpenGL
    // geometry for haptic rendering.
    hlBeginShape(HL_SHAPE_FEEDBACK_BUFFER, gHapticShapeId);

    // Use OpenGL commands to create geometry.
    beam1.DrawHL();//hay que "pintarlo" para poder tocarlo. Represento solo la
superficie que se puede contactar
        //beam2.DrawHL();

    // End the shape.
    hlEndShape();

    // End the haptic frame.
    hlEndFrame();
}


void drawCursor()
{
    static const double kCursorRadius = 0.5;
    static const double kCursorHeight = 1.5;
    static const int kCursorTess = 15;
    HLdouble proxyxform[16];

    GLUquadricObj *qobj = 0;

    glPushAttrib(GL_CURRENT_BIT | GL_ENABLE_BIT | GL_LIGHTING_BIT);
    glPushMatrix();

    if (!gCursorDisplayList)
    {
        gCursorDisplayList = glGenLists(1);
        glNewList(gCursorDisplayList, GL_COMPILE);
        qobj = gluNewQuadric();

        gluCylinder(qobj, 0.0, kCursorRadius, kCursorHeight,
                    kCursorTess, kCursorTess);
        glTranslated(0.0, 0.0, kCursorHeight);

        gluCylinder(qobj, kCursorRadius, 0.0, kCursorHeight / 5.0,
                    kCursorTess, kCursorTess);

        gluDeleteQuadric(qobj);
        glEndList();
    }

    // Get the proxy transform in world coordinates.
    hlGetDoublev(HL_PROXY_TRANSFORM, proxyxform);
    glMultMatrixd(proxyxform);
```

```
    // Apply the local cursor scale factor.
    glScaled(gCursorScale, gCursorScale, gCursorScale);

    glEnable(GL_COLOR_MATERIAL);
    glColor3f(0.0, 0.5, 1.0);

    glCallList(gCursorDisplayList);

    glPopMatrix();
    glPopAttrib();
}


void GestionaToque()
{
       int loadedNodeTouch = -1;
#if USING_API == 0
       hlGetDoublev(HL_DEVICE_POSITION, proxyPos);
       //GLdouble devicePosition[3];//punto de la superficie cargada mas proximo
al brazo haptico (no tiene porqué coincidir con un nodo ni es la posicion del brazo
haptico)

       Vector3<> proxy(proxyPos[0],proxyPos[1],proxyPos[2]);

       if (beam1.levelSet.InMesh(proxy))
       {
              double a = beam1.GetDistance(proxy); //Get the distance
              if (a < 0)
              {
                     loadedNodeTouch = beam1.GetClosestNode(proxy);
                     beam1.Deform(loadedNodeTouch, -a);
                     isTouching = true;
              }
              else
              {
                     isTouching = false;
              }
       }
       else
              DescargaOK(loadedNodeTouch);
#endif
#if USING_API == 1
       GLdouble devicePosition[3];
    // Proxy touching or not the shape.
    hlGetShapeBooleanv(gHapticShapeId, HL_PROXY_IS_TOUCHING, &isTouching);
       hlGetDoublev(HL_PROXY_POSITION, proxyPos);

    if (isTouching){
        loadedNodeTouch = beam1.GetClosestNode(Vector3<>(proxyPos[0], proxyPos[1],
proxyPos[2]));//nodo mas cercano al proxi
        if (loadedNodeTouch != -1){//por si acaso hay algun error

        //Opcion 1: uso InitialProxyPosition de la primera que toco vez como posi-
cion inicial para calcular el desplazamiento del brazo
        //calculo cada vez el devicePosition menos la InitialProxyPosition inicial
                     if (firstContact){
                            hlGetDoublev(HL_PROXY_POSITION, InitialProxyPosition);
                            firstContact = false;
                     }
            hlGetDoublev(HL_DEVICE_POSITION, devicePosition);
```

```
                    displacement    =    InitialProxyPosition[2]    -    devicePosi-
tion[2];//carga y desplazamiento en eje z

            if (displacement > 0){
                beam1.Deform(loadedNodeTouch, displacement);
            }
         }
         else Descarga(loadedNodeTouch);//toca un nodo que no pertenece al dominio
de las posibles posiciones de carga
     }
     else Descarga(loadedNodeTouch); //no toca el objeto
#endif

/*if(isTouching) std::cout << loadedNodeS << " isTouching" << displacement <<
std::endl;
else std::cout << loadedNodeS << std::endl;*/

}

void GestionaColision()
{
       int loadedNodeCollision = 0; //nota mental: NUNCA hay que usar variables
globales.

       Vector3<> punto = beam1.GetCoords(6);
       if (beam2.levelSet.InMesh(punto))
       {
              double a = beam2.GetDistance(punto); //Get the distance
              if (a < 0)
              {
                     loadedNodeCollision = beam2.GetClosestNode(punto);
                     beam2.Deform(loadedNodeCollision, -a);
                     isTouching = true;
              }
              else
              {
                     isTouching = false;
              }
       }
       else
              DescargaOK(loadedNodeCollision);

       //double dispArray[NumContact];
       //int loadedNodeArray[NumContact];

       //for (int i = 0; i < NumContact; i++)
       //{
       //     dispArray[i] = 0;
       //     loadedNodeArray[i] = 0;
       //     Vector3<> punto = beam1.GetCoords(nodosContacto[i]);
       //     if (beam2.levelSet.InMesh(punto))
       //     {
       //            double a = beam2.GetDistance(punto); //Get the distance
       //            loadedNodeArray[i] = beam2.GetClosestNode(punto);

       //            if (a < 0)
       //                   dispArray[i] = -a;
       //            else
       //                   dispArray[i] = 0;
       //     }
       //}
```

```
	//beam2.Deform(loadedNodeArray, dispArray);
}

void Descarga(int node)
{
//cuando ya no estamos tocando, la variable displacement debería ser cero, pero
esto produce golpes en el brazo
//debido a que entramos en una situación te toco-notoco-toco-notoco... Por eso se
va "descargando" poco a poco
//la variable displacement hasta dejarla en cero.
		firstContact = true;
	 if (displacement > 0){
	     displacement -= 0.01;//ajustado a mano
	 }
	 else{
	     displacement = 0;
	     //loadedNodeS = -1;
	 }
	 beam1.Deform(node, displacement);
}

void DescargaOK(int node)
{
	//beam1.Deform(node, 0);
	//beam2.Deform(node, 0);
}

void DrawAxis ()
{
	glBegin(GL_LINES);

	//Eje X
	GLfloat matAmbDiff2[] = { 1.0f, 0.0f, 0.0f, 1.0f };
	glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, matAmbDiff2);

	glVertex3f(0.0f, 0.0f, 0.0f);
	glVertex3f(3.0f, 0.0f, 0.0f);

	//Eje Y
	matAmbDiff2[0] = 0.0f;
	matAmbDiff2[1] = 1.0f;
	glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, matAmbDiff2);

	glVertex3f(0.0f, 0.0f, 0.0f);
	glVertex3f(0.0f, 3.0f, 0.0f);

	//Eje Z
	matAmbDiff2[1] = 0.0f;
	matAmbDiff2[2] = 1.0f;
	glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, matAmbDiff2);

	glVertex3f(0.0f, 0.0f, 0.0f);
	glVertex3f(0.0f, 0.0f, 3.0f);

	glEnd();
}
```

2 CLASE Solid

2.1 Cabecera

```
#pragma once
```

```
#include <stdlib.h>
#include <math.h>
#include <assert.h>
/*
#include <vector>
#include <iostream>
*/

#if defined(WIN32)
#include <windows.h>
#endif

#if defined(WIN32) || defined(linux)
#include <GL/glut.h>
#elif defined(__APPLE__)
#include <GLUT/glut.h>
#endif

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


#include <math.h>

#include "malla.h"
#include "parametros.h"
#include "Fs.h"
#include "Fx.h"
#include "Fy.h"
#include "Fz.h"

#include "Vector3.h"
#include "LSMesh.h"
#include "RotM.h"



class Solid {
	Vector3<> _centerPosition;
	Vector3<> _globalCoords[NumNodesSup]; //Coordenadas de los puntos respecto
a ejes globales
	Vector3<> _dispN[NumNodesSup]; //Desplazamientos de los nodos en un paso
dado
	RotM rotation;
	double _ScaleFactor;

public:
	LSMesh levelSet;
	//Constructores y destructor(es)
	Solid();
	Solid(Vector3<>, Vector3<>);
	Solid(double,double,double,double,double,double); //3 primeros argumentos:
coordenadas. 3 últimos argumentos: ángulos de rotación.
```

```
        void initializeGobalCoords(); //Copia las coordenadas de la malla a la va-
riable _globalCoords. Sólo se llama en los constructores.
        ~Solid();
        //GETs y SETs
        int GetClosestNode(Vector3<>); //Dado un punto en el espacio, se devuelve
el nodo de la superficie más cercano
        void SetCenter (double, double, double); // Las coordenadas deben ir respecto
a las globales
        void SetRotation (double, double, double); // Los ángulos han de ir en
radianes
        void SetCoords (int , Vector3<>);
        double GetLoad(); //Devuelve la carga que debe ir al brazo háptico según se
calcule en deform.
        Vector3<> GetCoords(int); //Devuelve un vector con la posición del punto.
        Vector3<> GetDisp(int); //Devuelve un vector con la posición del punto.

        double GetRefDisp(int);           //Deforma en Z la viga.
        //Movimientos como sólido rígido
        void Transform();
        void Translate(double, double, double);
        void Rotate(double, double, double);
        //Deformación del sólido
        void Deform(int, double); //Pasamos al nodo (int) la distancia (double), se
calcula la deformada y la fuerza a devolver.
        void Deform(int [], double []); //Overloaded function
        //Representación del sólido
        void Geometry();
        void DrawGL(float, float, float, float);//R G B y alpha
        void DrawHL(); //
        void DrawLS();
        //Otro
        Vector3<> GetNormal(int, int, int); //Dados tres enteros, devolvemos la
normal al triangulo que forman los nodos indicados por dichos enteros.
        double GetDistance(Vector3<>);

};
```

2.2 Cuerpo

```
#include "Solid.h"


//////////////////////////////////////////////////////
//                                  VERSIÓN 1.3
//
//////////////////////////////////////////////////////

/* ----------------------------------------
        SECCIÓN 1 - Constructores y Destructores
   ---------------------------------------- */
Solid :: Solid(){
        //initializeGobalCoords();

        /* Inicializamos el punto central y la orientación inicial del sólido a 0
*/
        SetCenter(0.0, 0.0, 0.0);
        SetRotation(0.0, 0.0, 0.0);
        Transform();
        levelSet.Transform(Vector3<>(0,0,0), Vector3<>(0,0,0));
}

Solid :: Solid(Vector3<> pos, Vector3<> ori){
        //initializeGobalCoords();
```

```
        /* Inicializamos el punto central */
        SetCenter(pos[0], pos[1], pos[2]);

        /* Inicializa la matriz de rotación */
        SetRotation(ori[0], ori[1], ori[2]);

        Transform();
        levelSet.Transform(pos, ori);

}

Solid :: Solid(double x, double y, double z, double a, double b, double c){
        //initializeGobalCoords();

        /* Inicializa el punto central */
        SetCenter(x, y, z);

        /* Inicializa la matriz de rotación  */
        SetRotation(a, b, c);

        Transform();
        levelSet.Transform(Vector3<>(x,y,z)
                                    , Vector3<>(a,b,c));
}

Solid :: ~Solid(){}

//void Solid :: initializeGobalCoords(){
//      for (int node = 0; node < NumNodesSup; node++){
//              _globalCoords[node].x = 0;
//      }
//}

/* ----------------------------------------
        SECCIÓN 2 - Métodos get y set:
   ---------------------------------------- */
void Solid :: SetCenter (double x, double y, double z) {
        /* Moidifica los valores del centro geométrico del sólido */
        _centerPosition = Vector3<>(x, y, z);
}

void Solid :: SetRotation (double a, double b, double c) {
        /* Modifica los valores de la matriz de rotación, según los ángulos de Euler
introducidos en radianes
        Ésta matriz es producto de 3 matrices, pero se define explícitamente
        Se rota primero en X, luego en Y y luego en Z */

        rotation.SetRotation(a,b,c);

}

void Solid :: SetCoords (int node, Vector3<> coor){
        _globalCoords[node] = coor;
}

double Solid :: GetLoad(){
        return _ScaleFactor;
}

Vector3<> Solid::GetNormal(int vertice1, int vertice2, int vertice3)
{
```

```
	//GLmanager necesita normales y vértices. Aquí se calcula la normal.
	//Este código es ultra guarro y espero poder mejorarlo en el futuro.

	/*	Tenemos los vértices del triángulo, que básicamente son coordenadas locales en la malla del sólido.
		Por tanto, hemos de cambiarlas a globales (incluyendo las deformaciones)	*/
	Vector3<> pointdef[3];
	pointdef[0] = _globalCoords[vertice1-1] + _dispN[vertice1-1];
	pointdef[1] = _globalCoords[vertice2-1] + _dispN[vertice2-1];
	pointdef[2] = _globalCoords[vertice3-1] + _dispN[vertice3-1];

	/*	Ahora lo que falta es dar dos aristas del triángulo en forma de vector y ya	*/
	Vector3<> u = pointdef[1] - pointdef[0];
	Vector3<> v = pointdef[2] - pointdef[0];

	return u^v;
}

Vector3<> Solid :: GetDisp(int node)
{
	return _dispN[node];
};

Vector3<> Solid :: GetCoords(int node)
{
	return (_dispN[node] + _globalCoords[node]);
}

int Solid :: GetClosestNode (Vector3<> proxy){
	double mindistance = proxy - _globalCoords[0];
	int loadedNodeS = 0;

	for (int node = 0; node < NumNodesS; node++) {
		Vector3<> aux = proxy - _globalCoords[LoadedNodesSup[node]];
		if (aux.GetModulus() < mindistance) {
			mindistance = aux.GetModulus();
			loadedNodeS = node;
		}
	}

	return loadedNodeS;
}

//Esto calcula el desplazamiento referencia de la barra, es decir el que ocurre con la carga de cálculo que se usó para generar
//el PGD.
double Solid::GetRefDisp(int nodo){
	double aux = 0;
	for (int jj = 0; jj < NumModosPGD; jj++)
		aux += Fz[jj][LoadedNodesSup[nodo]] * Fs[jj][nodo];

	return aux;
}

double Solid::GetDistance (Vector3<> proxy)
{
	return levelSet.DistanceToSurface(proxy);
}

/* ----------------------------------------
```

```
	SECCIÓN 3 - Métodos de sólido rígido:
		-transform
		-translate
		-rotate
   ----------------------------------------- */

void Solid::Transform() {
	/*	Ésta función actualiza las coordenadas globales de todos los puntos
del sólido como sólido rígido.
		En caso de que tengamos una traslación y una rotación simultáneas,
se recomienda usar
			setCenter(x, y, z);
			setRotation(a, b, c);
			transform();
		en vez de
			translate(x, y, z);
			rotate(a, b, c);
		para ahorrar cálculos.
	*/

	for (int node = 0; node < NumNodesSup; node++)
	{
		_globalCoords[node] = rotation.Rotation(Vector3<>(CoorSup[node][0],
CoorSup[node][1], CoorSup[node][2])) + _centerPosition;
	}
}

void Solid::Translate(double x, double y, double z) {
	/*	Mueve el objeto en el espacio
		NOTA: por ahora no compone (suma) traslaciones */

	SetCenter(x, y, z);
	Transform();
}

void Solid::Rotate(double a, double b, double c) {
	/*	Rota el objeto en el espacio
		NOTA: por ahora no compone (suma) rotaciones. */

	SetRotation(a, b, c);
	Transform();
}

/* -----------------------------------------
	SECCIÓN 4 - Métodos de deformación:
		-deform
		-DeformZ
   ----------------------------------------- */

/*	double Solid::SacarDeformacionEnZ(int nodo){
		displRef += Fz[jj][LoadedNodesSup[loadedNodeS]] * Fs[jj][loadedNo-
deS];
	}	*/

/*void Solid::Undeform(){

}*/

void Solid::Deform (int loadedNodeS, double disp){

	//Inicializar a 0
	for (int i = 0; i < NumNodesSup; i++){
```

```
		for (int j = 0; j < 3; j++){
			_dispN[i][j] = 0;
		}
	}

	//Comenzamos comprobando que no hay ningún error.
	if (loadedNodeS != -1){

		/*	Calculo la escala para que el desplazamiento del loadedNodeS coincida con el desplazamiento del brazo
		haptico desplazamiento para la carga dada en los ficheros de entrada */
	 double RefDisp = GetRefDisp(loadedNodeS); //Esto calcula el desplazamiento en Z del nodo cargado, con la carga con la que se hizo el precálculo

		/*	A continuación se compara con el desplazamiento de nuestro brazo (disp) para calcular el % de la carga
		original que habría que aplicar para conseguir una deformación "disp".*/
		if (RefDisp == 0) _ScaleFactor = 100;
		else _ScaleFactor = -(disp / RefDisp); // Creo que esto es el problema de los trompazos de la respuesta háptica. No puede ser problema de la API si estamos usando el LevelSet

		/* Se calculan los desplazamientos escalados de todos los nodos */
		for (int node = 0; node < NumNodesSup; node++) {
			for (int mode = 0; mode < NumModosPGD; mode++) {
				_dispN[node][0] += Fx[mode][node] * Fs[mode][loaded-NodeS] *_ScaleFactor;
				_dispN[node][1] += Fy[mode][node] * Fs[mode][loaded-NodeS] *_ScaleFactor;
				_dispN[node][2] += Fz[mode][node] * Fs[mode][loaded-NodeS] *_ScaleFactor;
			}
		}

		/*	Por último, se pasan las deformaciones a coordenadas globales
			*/
		for (int node = 0; node < NumNodesSup; node++){
			_dispN[node] = rotation.Rotation(_dispN[node]);
		}
	}
}

void Solid::Deform (int loadedNodeArray[NumContact], double dispArray[NumContact])
{
	//Inicializar a 0
	for (int i = 0; i < NumNodesSup; i++){
		for (int j = 0; j < 3; j++){
			_dispN[i][j] = 0;
		}
	}

	int tempLoad[NumContact];

	for (int i = 0; i < NumContact; i++)
	{
		//Comenzamos comprobando que no hay ningún error.
		if (loadedNodeArray[i] != -1){

			/*	Calculo la escala para que el desplazamiento del loa-dedNodeS coincida con el desplazamiento del brazo
```

```
                    haptico desplazamiento para la carga dada en los ficheros de
entrada */
                    double RefDisp = GetRefDisp(loadedNodeArray[i]); //Esto cal-
cula el desplazamiento en Z del nodo cargado, con la carga con la que se hizo el
precálculo

                    /*     A continuación se compara con el desplazamiento de nues-
tro brazo (disp) para calcular el % de la carga
                    original que habría que aplicar para conseguir una deformación
"disp".*/
                    if (RefDisp == 0) tempLoad[i] = 100;
                    else tempLoad[i] = -(dispArray[i] / RefDisp); // Creo que esto
es el problema de los trompazos de la respuesta háptica. No puede ser problema de
la API si estamos usando el LevelSet
                    _ScaleFactor += tempLoad[i] / NumContact;

                    /* Se calculan los desplazamientos escalados de todos los nodos
*/
                    for (int node = 0; node < NumNodesSup; node++) {
                           for (int mode = 0; mode < NumModosPGD; mode++) {

                                  _dispN[node][0]     +=     Fx[mode][node]     *
Fs[mode][loadedNodeArray[i]] *tempLoad[i] / NumContact;
                                  _dispN[node][1]     +=     Fy[mode][node]     *
Fs[mode][loadedNodeArray[i]] *tempLoad[i] / NumContact;
                                  _dispN[node][2]     +=     Fz[mode][node]     *
Fs[mode][loadedNodeArray[i]] *tempLoad[i] / NumContact;
                           }
                    }
             }
      }

      /*     Por último, se pasan las deformaciones a coordenadas globales
      */
      for (int node = 0; node < NumNodesSup; node++){
             _dispN[node] = rotation.Rotation(_dispN[node]);
      }
}



/* ----------------------------------------
      SECCIÓN 5 - Métodos de salida:
             -Geometry
             -drawGL
             -drawHL
   ---------------------------------------- */

void Solid::Geometry()
{
      Vector3<> pointdef[3];
      Vector3<> u, v, n;

      /* Geometría */
      glBegin(GL_TRIANGLES);
    for (GLint i_pol = 0; i_pol < NumElemsSup; i_pol++){//triangles

             for (GLint i_ver = 0; i_ver < 3; i_ver++){//vertices
                    GLint ver = ConnectSup[i_pol][i_ver]-1;
      //Creo que esto da la coordenada local del punto en la malla, por así
decirlo.
```

```
                        pointdef[i_ver]    =    _globalCoords[ver]    +    _dispN[ver];
        //Vale, se definen 3 puntos en coordenadas globales tras deformación.
                        /*     Básicamente todo lo anterior lo único que hace es cam-
biar las coordenadas del triángulo de "locales" a "globales" */
                }

                //calculate normal to the triangle
                u = pointdef[1] - pointdef[0];
                v = pointdef[2] - pointdef[0];
                n = u^v;            //Calculado como producto vectorial u x v.
                //Opción 2:
                //            n    =    (pointdef[1]    -    pointdef[0])^(pointdef[2]    -
pointdef[0]);
                //y pasamos de crear las variables u y v. Lo malo es que se ve menos
claro.

                //define triangle
                glNormal3f(n[0], n[1], n[2]);
                for (GLint i_ver = 0; i_ver < 3; i_ver++){
                        glVertex3f(pointdef[i_ver][0],            pointdef[i_ver][1],
pointdef[i_ver][2]);
                }
    }
    glEnd();
}

void Solid::DrawGL(float r, float g, float b, float alpha) {

        /* Definición del material */
         matGLfloatAmbDiff2[] = { r, g, b, alpha };     //Definimos el color y la
transparencia del sólido según la defina el usuario.
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, matAmbDiff2);

        /* Geometría */
        Geometry();

}


 void Solid::DrawHL(){
        Geometry();
 }

 //Dibujo del level set del sólido, únicamente para demostraciones o depuración
del código.
 void Solid::DrawLS() {

         int N = levelSet.GetDimension();

         /* Creamos Objeto GL */

         glPointSize(2.0f);

         glBegin(GL_POINTS);

         for (int i = 0; i < N; i++)
         {
                 if (levelSet.GetDistance(i) <= 0)
                 {
                        //Red if distance < 0
                        GLfloat matAmbDiff2[] = { 0.7, 0.1, 0.1, 1.0f };
```

```
                    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, mat-
AmbDiff2);
                }
                else
                {
                    //Green if distance > 0
                    GLfloat matAmbDiff2[] = { 0.1, 0.5, 0.1, 0.0 };
                    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, mat-
AmbDiff2);
                }
                Vector3<> punto = levelSet.GetVertex(i);
                glVertex3f(punto[0], punto[1], punto[2]);
               //std::cout << "Punto " << i << ": " << punto[i][0] << ", " <<
punto[i][0] << ", " << punto[i][2] << endl;
        }
        glEnd();
 }
```

2.3 Archivo malla.h

```
// malla.h
#if !defined(MALLA_H)
#define MALLA_H 1

// se definen dos mallas, la de superficie para vision y la de posiciones de carga
para sensación
const int NumNodesSup = 130;
const int NumElemsSup = 256;
const int NumNodesS = 48;
const int NumElemsS = 60;
const int NumContact = 9;
const double CoorSup[NumNodesSup][3] = {{1.500000, 0.100000, 0.100000}, {1.500000,
0.200000,  0.200000},  {0.000000,  0.100000,  0.200000},  {0.000000,  0.000000,
0.200000},  {0.300000,  0.000000,  0.100000},  {0.400000,  0.200000,  0.000000},
{1.500000,  0.100000,  0.000000},  {0.300000,  0.100000,  0.200000},  {0.100000,
0.000000,  0.100000},  {0.100000,  0.100000,  0.000000},  {0.500000,  0.100000,
0.200000},  {0.600000,  0.200000,  0.000000},  {0.200000,  0.100000,  0.200000},
{0.100000,  0.200000,  0.000000},  {0.000000,  0.200000,  0.100000},  {1.000000,
0.200000,  0.000000},  {0.900000,  0.100000,  0.200000},  {1.000000,  0.000000,
0.200000},  {1.000000,  0.100000,  0.200000},  {1.100000,  0.200000,  0.200000},
{1.100000,  0.200000,  0.100000},  {0.800000,  0.200000,  0.200000},  {1.000000,
0.200000,  0.200000},  {1.500000,  0.200000,  0.000000},  {1.400000,  0.100000,
0.000000},  {0.700000,  0.200000,  0.000000},  {0.200000,  0.100000,  0.000000},
{1.200000,  0.100000,  0.000000},  {0.100000,  0.200000,  0.200000},  {1.000000,
0.000000,  0.100000},  {1.100000,  0.000000,  0.000000},  {1.200000,  0.200000,
0.100000},  {0.100000,  0.100000,  0.200000},  {1.500000,  0.000000,  0.100000},
{0.100000,  0.000000,  0.000000},  {0.100000,  0.000000,  0.200000},  {1.200000,
0.000000,  0.000000},  {1.200000,  0.100000,  0.200000},  {1.200000,  0.000000,
0.100000},  {1.300000,  0.000000,  0.200000},  {1.200000,  0.000000,  0.200000},
{0.300000,  0.000000,  0.200000},  {0.300000,  0.200000,  0.100000},  {0.200000,
0.000000,  0.000000},  {0.300000,  0.200000,  0.200000},  {0.200000,  0.000000,
0.200000},  {0.900000,  0.000000,  0.000000},  {0.400000,  0.000000,  0.000000},
{0.700000,  0.200000,  0.200000},  {0.500000,  0.000000,  0.200000},  {0.400000,
0.100000,  0.000000},  {0.500000,  0.200000,  0.100000},  {0.400000,  0.100000,
0.200000},  {0.100000,  0.200000,  0.100000},  {0.000000,  0.200000,  0.200000},
{0.200000,  0.200000,  0.200000},  {1.500000,  0.100000,  0.200000},  {1.500000,
0.000000,  0.200000},  {1.400000,  0.200000,  0.000000},  {1.300000,  0.100000,
0.200000},  {1.400000,  0.100000,  0.200000},  {1.400000,  0.000000,  0.000000},
{0.800000,  0.000000,  0.000000},  {0.700000,  0.000000,  0.100000},  {0.800000,
0.000000,  0.200000},  {0.900000,  0.200000,  0.200000},  {0.700000,  0.100000,
0.200000},  {0.700000,  0.000000,  0.000000},  {0.800000,  0.100000,  0.200000},
{1.100000,  0.100000,  0.000000},  {0.800000,  0.000000,  0.100000},  {1.100000,
0.100000,  0.200000},  {1.400000,  0.200000,  0.200000},  {1.500000,  0.000000,
```

```
0.000000},  {0.000000,  0.100000,  0.100000},  {0.300000,  0.100000,  0.000000},
{0.300000,  0.000000,  0.000000},  {0.400000,  0.000000,  0.200000},  {0.600000,
0.000000,  0.100000},  {0.600000,  0.100000,  0.200000},  {0.500000,  0.000000,
0.100000},  {0.800000,  0.200000,  0.000000},  {0.300000,  0.200000,  0.000000},
{0.700000,  0.000000,  0.200000},  {0.900000,  0.200000,  0.000000},  {0.800000,
0.100000,  0.000000},  {0.200000,  0.200000,  0.100000},  {0.900000,  0.100000,
0.000000},  {0.500000,  0.200000,  0.200000},  {0.700000,  0.200000,  0.100000},
{1.000000,  0.100000,  0.000000},  {0.000000,  0.100000,  0.000000},  {0.900000,
0.000000,  0.100000},  {1.300000,  0.200000,  0.200000},  {1.400000,  0.000000,
0.100000},  {1.300000,  0.000000,  0.100000},  {1.400000,  0.000000,  0.200000},
{1.100000,  0.000000,  0.100000},  {1.100000,  0.000000,  0.200000},  {0.000000,
0.200000,  0.000000},  {0.500000,  0.000000,  0.000000},  {0.400000,  0.000000,
0.100000},  {0.500000,  0.100000,  0.000000},  {1.500000,  0.200000,  0.100000},
{1.300000,  0.000000,  0.000000},  {1.300000,  0.200000,  0.100000},  {1.200000,
0.200000,  0.000000},  {1.300000,  0.100000,  0.000000},  {1.000000,  0.200000,
0.100000},  {0.500000,  0.200000,  0.000000},  {0.200000,  0.200000,  0.000000},
{0.200000,  0.000000,  0.100000},  {1.000000,  0.000000,  0.000000},  {1.200000,
0.200000,  0.200000},  {0.600000,  0.100000,  0.000000},  {1.300000,  0.200000,
0.000000},  {1.100000,  0.200000,  0.000000},  {0.800000,  0.200000,  0.100000},
{0.600000,  0.200000,  0.200000},  {0.400000,  0.200000,  0.100000},  {0.900000,
0.200000,  0.100000},  {0.600000,  0.200000,  0.100000},  {0.600000,  0.000000,
0.000000},  {0.000000,  0.000000,  0.100000},  {0.400000,  0.200000,  0.200000},
{0.000000,  0.000000,  0.000000},  {0.900000,  0.000000,  0.200000},  {0.700000,
0.100000,  0.000000},  {0.600000,  0.000000,  0.200000},  {1.400000,  0.200000,
0.100000}};

const int ConnectSup[NumElemsSup][3] = {{1, 2, 57}, {2, 104, 73}, {3, 4, 33}, {4,
124, 36}, {5, 78, 42}, {6, 51, 76}, {7, 62, 25}, {8, 125, 45}, {8, 53, 125}, {9,
126, 35}, {10, 126, 92}, {11, 80, 119}, {11, 119, 89}, {12, 115, 103}, {12, 110,
52}, {13, 56, 33}, {14, 100, 15}, {15, 100, 75}, {16, 121, 109}, {17, 23, 66}, {17,
19, 23}, {18, 93, 30}, {19, 127, 18}, {20, 19, 72}, {21, 23, 20}, {22, 121, 118},
{23, 21, 109}, {24, 104, 7}, {25, 59, 24}, {26, 128, 115}, {26, 12, 122}, {27, 10,
111}, {28, 70, 107}, {29, 54, 55}, {20, 72, 114}, {21, 117, 109}, {30, 99, 18},
{31, 113, 70}, {31, 98, 113}, {32, 21, 20}, {33, 29, 3}, {7, 1, 74}, {34, 62, 74},
{13, 36, 46}, {35, 10, 27}, {36, 13, 33}, {37, 31, 28}, {37, 39, 31}, {38, 60, 94},
{39, 96, 40}, {40, 60, 41}, {28, 31, 70}, {41, 60, 38}, {42, 46, 112}, {5, 44, 77},
{43, 56, 45}, {8, 46, 42}, {44, 27, 76}, {42, 112, 5}, {45, 13, 8}, {46, 8, 13},
{47, 63, 88}, {47, 93, 63}, {17, 65, 127}, {6, 120, 110}, {16, 85, 121}, {48, 103,
101}, {49, 80, 67}, {50, 11, 78}, {51, 48, 77}, {52, 125, 89}, {53, 78, 11}, {53,
11, 89}, {54, 14, 15}, {54, 15, 55}, {55, 3, 29}, {27, 44, 35}, {15, 75, 3}, {56,
29, 33}, {22, 69, 66}, {57, 58, 34}, {58, 57, 97}, {34, 1, 57}, {59, 116, 106},
{60, 61, 73}, {61, 40, 97}, {10, 14, 111}, {59, 25, 108}, {62, 95, 105}, {63, 68,
86}, {63, 71, 68}, {64, 71, 65}, {65, 69, 84}, {66, 121, 22}, {67, 22, 49}, {67,
69, 22}, {67, 129, 84}, {42, 78, 53}, {52, 120, 125}, {49, 119, 80}, {68, 123,
128}, {68, 64, 123}, {69, 67, 84}, {70, 113, 91}, {17, 127, 19}, {69, 17, 66}, {71,
127, 65}, {72, 19, 18}, {18, 99, 72}, {38, 94, 114}, {61, 60, 40}, {73, 130, 94},
{2, 73, 61}, {74, 62, 7}, {57, 61, 97}, {35, 44, 112}, {75, 4, 3}, {36, 124, 9},
{55, 15, 3}, {75, 124, 4}, {76, 83, 6}, {4, 36, 33}, {77, 44, 76}, {78, 102, 50},
{79, 81, 101}, {50, 102, 81}, {80, 50, 129}, {5, 102, 78}, {79, 129, 81}, {81, 129,
50}, {79, 84, 129}, {79, 123, 64}, {82, 85, 86}, {83, 111, 87}, {64, 84, 79}, {80,
129, 67}, {84, 64, 65}, {85, 82, 118}, {86, 68, 128}, {29, 56, 87}, {87, 54, 29},
{88, 63, 86}, {18, 127, 93}, {82, 26, 90}, {89, 122, 52}, {90, 119, 49}, {83, 87,
43}, {91, 47, 88}, {30, 98, 99}, {72, 99, 38}, {92, 126, 124}, {39, 41, 98}, {39,
98, 31}, {41, 99, 98}, {93, 127, 71}, {30, 93, 47}, {94, 130, 106}, {41, 39, 40},
{95, 34, 58}, {96, 97, 40}, {96, 105, 95}, {97, 95, 58}, {57, 2, 61}, {95, 97, 96},
{1, 7, 104}, {37, 108, 105}, {98, 30, 113}, {28, 108, 37}, {99, 41, 38}, {100, 92,
75}, {81, 102, 48}, {101, 115, 123}, {77, 48, 102}, {101, 81, 48}, {102, 5, 77},
{103, 48, 51}, {77, 76, 51}, {92, 124, 75}, {11, 50, 80}, {104, 130, 73}, {69, 65,
17}, {7, 25, 24}, {105, 96, 37}, {94, 60, 73}, {106, 116, 32}, {32, 107, 21}, {37,
96, 39}, {43, 87, 56}, {106, 130, 59}, {24, 130, 104}, {107, 116, 28}, {107, 32,
116}, {108, 28, 116}, {88, 16, 91}, {93, 71, 63}, {71, 64, 68}, {43, 120, 6}, {109,
66, 23}, {110, 51, 6}, {90, 26, 122}, {82, 86, 128}, {103, 115, 101}, {46, 9, 112},
```

```
{46, 36, 9}, {45, 56, 13}, {9, 124, 126}, {43, 45, 120}, {76, 27, 83}, {111, 54,
87}, {14, 92, 100}, {35, 126, 10}, {14, 10, 92}, {24, 59, 130}, {108, 25, 105},
{70, 91, 117}, {85, 88, 86}, {53, 8, 42}, {85, 16, 88}, {106, 114, 94}, {52, 110,
120}, {103, 51, 110}, {111, 83, 27}, {111, 14, 54}, {112, 9, 35}, {43, 6, 83}, {5,
112, 44}, {16, 117, 91}, {16, 109, 117}, {113, 47, 91}, {113, 30, 47}, {114, 106,
32}, {115, 128, 123}, {26, 82, 128}, {114, 32, 20}, {116, 59, 108}, {20, 23, 19},
{90, 118, 82}, {117, 107, 70}, {117, 21, 107}, {105, 25, 62}, {34, 74, 1}, {12,
26, 115}, {118, 49, 22}, {119, 90, 122}, {1, 104, 2}, {72, 38, 114}, {120, 45,
125}, {121, 66, 109}, {110, 12, 103}, {118, 121, 85}, {122, 12, 52}, {90, 49, 118},
{123, 79, 101}, {122, 89, 119}, {89, 125, 53}, {34, 95, 62}};

const double CoorS[NumNodesS][3] = {{0.000000, 0.000000, 0.200000}, {0.000000,
0.200000, 0.200000}, {1.500000, 0.200000, 0.200000}, {1.500000, 0.000000,
0.200000}, {0.000000, 0.100000, 0.200000}, {0.100000, 0.200000, 0.200000},
{0.200000, 0.200000, 0.200000}, {0.300000, 0.200000, 0.200000}, {0.400000,
0.200000, 0.200000}, {0.500000, 0.200000, 0.200000}, {0.600000, 0.200000,
0.200000}, {0.700000, 0.200000, 0.200000}, {0.800000, 0.200000, 0.200000},
{0.900000, 0.200000, 0.200000}, {1.000000, 0.200000, 0.200000}, {1.100000,
0.200000, 0.200000}, {1.200000, 0.200000, 0.200000}, {1.300000, 0.200000,
0.200000}, {1.400000, 0.200000, 0.200000}, {1.400000, 0.000000, 0.200000},
{1.300000, 0.000000, 0.200000}, {1.200000, 0.000000, 0.200000}, {1.100000,
0.000000, 0.200000}, {1.000000, 0.000000, 0.200000}, {0.900000, 0.000000,
0.200000}, {0.800000, 0.000000, 0.200000}, {0.700000, 0.000000, 0.200000},
{0.600000, 0.000000, 0.200000}, {0.500000, 0.000000, 0.200000}, {0.400000,
0.000000, 0.200000}, {0.300000, 0.000000, 0.200000}, {0.200000, 0.000000,
0.200000}, {0.100000, 0.000000, 0.200000}, {1.500000, 0.100000, 0.200000},
{1.400000, 0.100000, 0.200000}, {1.300000, 0.100000, 0.200000}, {1.200000,
0.100000, 0.200000}, {1.100000, 0.100000, 0.200000}, {1.000000, 0.100000,
0.200000}, {0.900000, 0.100000, 0.200000}, {0.800000, 0.100000, 0.200000},
{0.700000, 0.100000, 0.200000}, {0.600000, 0.100000, 0.200000}, {0.500000,
0.100000, 0.200000}, {0.400000, 0.100000, 0.200000}, {0.300000, 0.100000,
0.200000}, {0.200000, 0.100000, 0.200000}, {0.100000, 0.100000, 0.200000}};

const int ConnectS[NumElemsS][3] = {{12, 11, 42}, {5, 33, 48}, {10, 9, 44}, {39,
15, 14}, {37, 21, 36}, {17, 36, 18}, {24, 39, 40}, {36, 35, 18}, {37, 22, 21}, {21,
20, 36}, {20, 4, 35}, {44, 45, 29}, {43, 10, 44}, {44, 29, 28}, {45, 46, 30}, {47,
6, 48}, {27, 43, 28}, {9, 8, 45}, {32, 48, 33}, {8, 46, 45}, {46, 31, 30}, {46,
47, 31}, {33, 5, 1}, {48, 2, 5}, {32, 47, 48}, {32, 31, 47}, {48, 6, 2}, {47, 7,
6}, {47, 46, 7}, {7, 46, 8}, {28, 43, 44}, {29, 45, 30}, {44, 9, 45}, {42, 43, 27},
{11, 10, 43}, {42, 11, 43}, {26, 41, 42}, {40, 39, 14}, {41, 13, 12}, {42, 41, 12},
{40, 14, 13}, {35, 19, 18}, {35, 34, 19}, {38, 16, 15}, {37, 36, 17}, {16, 38, 37},
{19, 34, 3}, {35, 4, 34}, {23, 22, 38}, {16, 37, 17}, {38, 22, 37}, {27, 26, 42},
{40, 13, 41}, {41, 26, 25}, {38, 15, 39}, {40, 41, 25}, {40, 25, 24}, {39, 24, 23},
{39, 23, 38}, {36, 20, 35}};

const int LoadedNodesSup[NumNodesS] = {3, 54, 1, 57, 2, 28, 55, 44, 124, 88, 118,
48, 21, 65, 22, 19, 113, 93, 72, 96, 39, 40, 98, 17, 126, 64, 83, 128, 49, 77, 41,
45, 35, 56, 60, 59, 37, 71, 18, 16, 68, 66, 79, 10, 52, 7, 12, 32};

const int nodosContacto[NumContact] = {143, 135, 167, 120, 104, 145, 9, 147, 12};

#endif
```

3 CLASE LSMesh

3.1 Cabecera

```
#pragma once

#include <cmath>
#include "Vector3.h"
#include "LSData.h"
```

```
#include "RotM.h"
#include <vector>

using namespace std;

class LSMesh
{
	Vector3<> vertices[Nx*Ny*Nz];
	RotM rotation;
	Vector3<> translation;
	Vector3<> D;
public:
	//Constructors
	LSMesh();
	//Default constructor. It takes the data directly from LSdata.h. This con-
structor is intended for use in the real time simulation.
	//Get & Set
	double GetDistance(int, int, int);				//Returns the
stored distance of the (i,j,k) point of the level set.
	double GetDistance(int);					//Returns the
stored distance of the (i) point of the level set.
	int GetDimension();
	Vector3<> GetVertex(int, int, int);
	Vector3<> GetVertex(int);
	int GetLoadStates();
	//void SetDistance(int, int, double);				//
	//Other
	Vector3<int> N2Coor(int);
	//Transforms 1 integer into the (i,j,k) 3D location
	int Coor2N(int, int, int);
	//Transforms 3 integers (i,j,k), into the location in the array m
	Vector3<int> WCoor2TCoor(double, double, double);	//Transforms (x,y,z)
world coordinates into (i,j,k) tensor coordinates
	Vector3<> TCoor2WCoor(Vector3<int>);			//Transforms
(i,j,k) tensor coordinates into (x,y,z) world coordinates
	bool InMesh (Vector3<>);			//Tells us if a point is in-
side the mesh. If it's inside, it puts the 8 nodes in the array of Vector3
	//bool InMesh (Vector3<>, double&);
	//Tells us if a point is inside the mesh. If it's inside, it gives us the
interpolated distance to the surface.
	void Transform (Vector3<>, Vector3<>);			//Uh	summa
lumma dooma lumma you assuming I'm a human What I gotta do to get it through to
you I'm superhuman
	double DistanceToSurface(Vector3<>);
};
```

3.2 Cuerpo

```
#include "LSMesh.h"

//VERSION
#define VER 2

//////////////CONSTRUCTORS/////////////

LSMesh::LSMesh()
{
	for (int i=0; i<Nx*Ny*Nz ; i++)
	{
		for (int j= 0; j<3 ; j++)
		{
			vertices[i][j] = 0;
		}
```

```
	}

	D = Vector3<>(Dx, Dy, Dz);
}

//////////////////GET&SET//////////////////

double LSMesh::GetDistance(int i)
{
	return LSMatrix[i];
}

double LSMesh::GetDistance(int i, int j, int k)
{
	return LSMatrix[Coor2N(i,j,k)];
}

Vector3<> LSMesh::GetVertex(int i, int j, int k)
{
	return vertices[Coor2N(i,j,k)];
}

Vector3<> LSMesh::GetVertex(int n)
{
	return vertices[n];
}

int LSMesh::GetDimension(){
	return Nx*Ny*Nz;
}

//////////////////OTHER//////////////////////

//	Since we store a three-dimensional mesh into a one-dimensional array,
//	we need this function to transform the position of the array into a
//	set of 3 integers (which work as "in-mesh coordinates").
Vector3<int> LSMesh::N2Coor(int n){

	Vector3<int> answer;
	////////////////////////
	int i = n % Nx;
	int aux = (n-i)/Nx; //integer division
	int j = aux % Ny;
	int k = (aux - j)/Ny;
	////////////////////////
	answer[0] = i; answer[1] = j; answer[2] = k;
	return answer;
}

//	This fucntion is the inverse of the previous one,
//	i.e. transforms "in-mesh coordinates" into an actual array location
//	(or basically, mimics a 3-D array allocation).
int LSMesh::Coor2N(int i, int j, int k){
	return i + j*Nx + k*Nx*Ny;
}

//	Transforms WORLD coordinates (i.e. a point in euclidean space)
//	into MESH coordinates (i.e. the indexes that point would have in the array)
Vector3<int> LSMesh::WCoor2TCoor(double x, double  y, double  z){
	Vector3<int> answer(int(x / Dx), int(y / Dy), int(z / Dz));
	return answer;
}
```

```
//	Inverse of the previous function.
Vector3<> LSMesh::TCoor2WCoor(Vector3<int> intVector){
	Vector3<> v(intVector[0]*Dx, intVector[1]*Dy, intVector[2]*Dz);
	return v;
}

//Returns true if a point is inside the whole mesh. It also stores the indices of
the LSVertices engulfing the point.
bool LSMesh::InMesh (Vector3<> point)
{
	point = rotation.InvRotation(point - translation);
	//Basically, if the point x coordinate is between the x coordinate of the
vertice (0,0,0) and (Nx,0,0),
	//and the same occurs for y and z, then we consider that the point is INSIDE
the mesh.
	return (LSVertices[0][0] < point[0] && point[0] < LSVertices[Coor2N(Nx-
1,0,0)][0]) &&
			(LSVertices[0][1] < point[1] && point[1] < LSVerti-
ces[Coor2N(0,Ny-1,0)][1]) &&
			(LSVertices[0][2] < point[2] && point[2] < LSVerti-
ces[Coor2N(0,0,Nz-1)][2]);
}

double LSMesh::DistanceToSurface(Vector3<> point)
{
	point = rotation.InvRotation(point - vertices[0]);
	//point -= Vector3<>(LSVertices[0][0], LSVertices[0][1], LSVertices[0][2]);
	//point -= vertices[0];

	int i = int (point[0] / D[0]); //Ahora i es la parte entera de la división.
	int j = int (point[1] / D[1]);
	int k = int (point[2] / D[2]);

	double p = 2*(((point[0] / D[0]) - i)-0.5);
	double q = 2*(((point[1] / D[1]) - j)-0.5);
	double r = 2*(((point[2] / D[2]) - k)-0.5);

	int  xi[8] = {-1, 1, 1, -1, -1, 1, 1, -1};
	int eta[8] = {-1, -1, 1, 1, -1, -1, 1, 1};
	int  mu[8] = {-1, -1, -1, -1, 1, 1, 1, 1};

	int iff[8][3] = {{0,0,0}, {0,0,1}, {0,1,1}, {0,1,0}, {1,0,0}, {1,0,1},
{1,1,1}, {1,1,0}};

	double answer = 0;

	for (int h = 0; h < 8; h++)
	{
		answer += GetDistance(i+iff[h][0], j+iff[h][1],
k+iff[h][2])*0.125*(1+p*xi[h])*(1+q*eta[h])*(1+r*mu[h]);
	}

	return answer;
}

void LSMesh::Transform(Vector3<> translation, Vector3<> rot)
{
	//Orientation matrix
	rotation.SetRotation(rot[0], rot[1], rot[2]);
	this->translation = translation;
```

```
	for (int node = 0; node < GetDimension(); node++){
		vertices[node]	=	rotation.Rotation(Vector3<>(LSVertices[node][0],LSVertices[node][1],LSVertices[node][2])) + translation;
	}

	D = rotation.Rotation(D);
}
```

4 ARCHIVO Vector3

4.1 Cabecera

```
//version 1.4
#pragma once

#include <cmath>

template <class T = double>
class Vector3 {

public:
	T x;
	T y;
	T z;

	//Constructors
	Vector3<T>();
	Vector3<T>(T, T, T);
	Vector3<T>(const Vector3<T> &);

	//Overloaded operators
	Vector3<T>&	operator= (const Vector3<T> &);		//Asignation
	Vector3<T>	operator+ (const Vector3<T> &);		//Sum
	Vector3<T>	operator- (const Vector3<T> &);		//Rest
	T			operator* (const Vector3<T> &);		//Dot product
	Vector3<T>	operator^ (const Vector3<T> &);		//Cross product


	Vector3<T>	operator+= (const Vector3<T> &);	//Equal Plus
	Vector3<T>	operator-= (const Vector3<T> &);	//Equal Minus
	bool		operator== (const Vector3<T> &);	//Equal to
	bool		operator!= (const Vector3<T> &);	//Not equal to

	T&			operator[] (const int &);			//Subscript
operator
	T			operator[] (const int &) const;		//Read-only
subscript operator

	//Other functions
	double GetModulus();
	void Copy(const Vector3<T> &);
};

#include "Vector3.cpp"
```

4.2 Cuerpo

```
//version 1.4

#include "Vector3.h"

///////////////CONSTRUCTORS///////////////
template <class T>
```

```
Vector3<T>::Vector3()
{
	x=0;
	y=0;
	z=0;
}

template <class T>
Vector3<T>::Vector3 (T a, T b, T c)
{
	x = a;
	y = b;
	z = c;
}

template <class T>
Vector3<T>::Vector3 (const Vector3<T> &c)
{
	Copy(c);
}

////////////OPERATOR OVERLOAD////////////
template <class T>
Vector3<T>&
Vector3<T>::operator= (const Vector3<T> &v)
{
	if (this != &v)
	{
		(*this).~Vector3<T>();	//Releases the memory previously occupied
by this vector3

		Copy(v);
	}

	return *this;
}

template <class T>
Vector3<T>
Vector3<T>::operator+ (const Vector3<T> &v)
{
	Vector3<T> temp;

	temp.x = x + v.x;
	temp.y = y + v.y;
	temp.z = z + v.z;

	return temp;
}

template <class T>
Vector3<T>
Vector3<T>::operator- (const Vector3<T> &v)
{
	Vector3<T> temp;

	temp.x = x - v.x;
	temp.y = y - v.y;
	temp.z = z - v.z;

	return temp;
}
```

```
template <class T>
T
Vector3<T>::operator* (const Vector3<T> &v)
{
	return x*v.x + y*v.y + z*v.z;
}

template <class T>
Vector3<T> Vector3<T>::operator^ (const Vector3<T> &v)
{
	Vector3<T> temp;

	temp.x = y*v.z - z*v.y;
	temp.y = z*v.x - x*v.z;
	temp.z = x*v.y - y*v.x;

	return temp;
}



template <class T>
Vector3<T>
Vector3<T>::operator+= (const Vector3<T> &v)
{
	x += v.x;
	y += v.y;
	z += v.z;

	return *this;
}

template <class T>
Vector3<T>
Vector3<T>::operator-= (const Vector3<T> &v)
{
	x -= v.x;
	y -= v.y;
	z -= v.z;

	return *this;
}


template <class T>
bool
Vector3<T>::operator== (const Vector3<T> &v)
{

	return (x == v.x && y == v.y && z == v.z);
}

template <class T>
bool
Vector3<T>::operator!= (const Vector3<T> &v)
{

	return !(*this == v);
}

template <class T>
```

```
T&
Vector3<T>::operator[] (const int &nIndex)
{
	if (nIndex < 1) return x;
	if (nIndex == 1) return y;
	if (nIndex > 1) return z;
}

template <class T>
T
Vector3<T>::operator[] (const int &nIndex) const
{
	if (nIndex < 1) return x;
	if (nIndex == 1) return y;
	if (nIndex > 1) return z;
}

//////////////////OTHER/////////////////
template <class T>
double
Vector3<T>::GetModulus()
{
	Vector3<T> temp (x,y,z);

	return sqrt(temp*temp);
}

template <class T>
void
Vector3<T>::Copy(const Vector3<T> &c)
{
	x = c.x;
	y = c.y;
	z = c.z;
}
```

5 CLASE RotM

5.1 Cabecera

```
//version 1.4
#pragma once

#include "Vector3.h"
#include <cmath>

class RotM {
	double comp[3][3];
	double inv[3][3];

public:
	//Constructors
	RotM();
	RotM(double, double, double);

	//Other functions
	Vector3<> Rotation(Vector3<> );
	Vector3<> InvRotation(Vector3<> );
	void SetRotation(double, double, double);
};
```

5.2 Cuerpo

```
#include "RotM.h"

//CONSTRUCTORS
RotM::RotM()
{
	double aux = 0;

	for (int row = 0; row < 3; row++)
	{
		for (int col = 0; col < 3; col++)
		{
			if (row == col) aux = 1;
			comp[row][col] = aux;
			inv[row][col] = aux;
			aux = 0;
		}
	}
}

RotM::RotM(double a, double b, double c)
{
	SetRotation(a,b,c);
}

//FUNCTIONS
Vector3<> RotM::Rotation(Vector3<> c)
{
	Vector3<> ans;

	for (int i = 0; i < 3; i++)
	{
		for (int j = 0; j < 3; j++)
		{
			ans[i] += comp[i][j] * c[j];
		}
	}

	return ans;
}

Vector3<> RotM::InvRotation(Vector3<> c)
{
	Vector3<> ans;

	for (int i = 0; i < 3; i++)
	{
		for (int j = 0; j < 3; j++)
		{
			ans[i] += inv[i][j] * c[j];
		}
	}

	return ans;
}

void RotM::SetRotation(double a, double b, double c)
{
	comp[0][0] = cos(b)*cos(c);
	comp[0][1] = -cos(b)*sin(c);
	comp[0][2] =  sin(b);
```

```
       comp[1][0] = cos(a)*sin(c) + sin(a)*sin(b)*cos(c);    comp[1][1]         =
cos(a)*cos(c) - sin(a)*sin(b)*sin(c);    comp[1][2] = -sin(a)*cos(b);
       comp[2][0] = sin(a)*sin(c) - cos(a)*sin(b)*cos(c);    comp[2][1]         =
sin(a)*cos(c) + cos(a)*sin(b)*sin(c);    comp[2][2] =  cos(a)*cos(b);

       inv[0][0] = cos(b)*cos(c);
       inv[0][1] = cos(a)*sin(c)+sin(a)*sin(b)*cos(c);       inv[0][2]          =
sin(a)*sin(c)-cos(a)*sin(b)*cos(c);
       inv[1][0] = -cos(b)*sin(c);
       inv[1][1] = cos(a)*cos(c)-sin(a)*sin(b)*sin(c);       inv[1][2]          =
cos(a)*sin(b)*sin(c)+sin(a)*cos(c);
       inv[2][0] = sin(b);
       inv[2][1] = -sin(a)*cos(b);
       inv[2][2] = cos(a)*cos(b);
}
```

ANEXO C

CÓDIGO DE LA CALCULADORA DE DISTANCE FIELD

1 ARCHIVO LevelSet Calculator.cpp

```
#include <fstream>
#include <iostream>
#include <cmath>
#include "LS\LSMesh.h"
#include "Vector3\Vector3.h"
#include "Solid\Solid.h"

#define DEBUG              0
#define INSIDE_OUTSIDE     1
#define SHOW               0

using namespace std;

/*El level set va a ser una nube de puntos con información sobre la distancia a
una geometría, si el centro de la nube
  y el de la geometría coincidieran.

  Éste hecho hace que para su aplicación, los datos de Level Set tengan que ser
rotados y trasladados con la misma rotación y traslación
  que tenga el solido */

/* FUNCTION PROTOTYPES */
void WriteFile(LSMesh);                  //Creates a .h header with the needed
data.
int InOut(Vector3<>, Vector3<>);  //Returns -1 if both vectors point in the same
direction, 1 if they have opposite directions.
void Calculate(LSMesh*, Solid);
void GetData(); //Gets the level set parameters.

/* GLOBAL VARIABLES */
//Modify these to change the LSMesh parameters.
       double _Dx    = .1;
       double _Dy    = .1;
       double _Dz    = .1;
       int _Nx             = 16;
       int _Ny             = 4;
       int _Nz             = 9;
       Vector3<> translation(0.0, -0.05, -0.55); //x, y, z

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

```
	#if DEBUG == 0
	cout << "baking..." << endl;
	#endif
	//GetData(); // For an hypotetical final release, console app.

	LSMesh mesh(_Dx, _Dy, _Dz, _Nx, _Ny, _Nz);
	Solid beam;

	Calculate(&mesh, beam); //Se hace un cáculo con la viga indeformada

	#if DEBUG == 1
		char end;
		int k = 7;
		int j = 1;

		cout << mesh.GetDistance(0, j, k) << endl;
		cout << mesh.GetDistance(1, j, k) << endl;
		cout << mesh.GetDistance(2, j, k) << endl;
		cout << mesh.GetDistance(3, j, k) << endl;
		cin >> end;
	#endif

	#if DEBUG == 0
	WriteFile(mesh);
	#endif

	#if SHOW == 1
	beam.DrawGL();
	beam.DrawLS();
	#endif
}

void Calculate(LSMesh *mesh, Solid beam){
	int s;

	for (int d = 0; d < mesh->GetDimension(); d++) {

		Vector3<> p = mesh->TCoor2WCoor(mesh->N2Coor(d)) + translation;
//Coordenadas espaciales del nodo del Level Set del que toca calcular la distancia
d.
		s = beam.GetClosestNode(p); //Nodo s del sólido más cercano a p (claro
que aquí ya estamos calculando distancia euclídea entre aux y los nodos del só-
lido...)
		Vector3<> sVector = beam.GetCoords(s); //Creamos un vector que re-
presenta s en el espacio tridimensional.
		Vector3<> v = p - sVector; //Vector que une p con s
		int fuera = 0;

		#if INSIDE_OUTSIDE == 1
		// Detección dentro / fuera.

		//Para el nodo s del sólido, buscamos todos los posibles triángulos
en los que esté
		//for (int i = 0; i < NumElemsSup && fuera != 1; i++) {
	//Recorre todos los triángulos
		//	for (int j = 0; j < 3; j++){		//Recorre los vérti-
ces de dicho triángulo
		//		if (ConnectSup[i][j]-1 == s){	//Si uno de los vér-
tices es el nodo...
		//			Vector3<> n;
```

```
//                  n   =     beam.GetNormal(ConnectSup[i][0],  Con-
nectSup[i][1], ConnectSup[i][2]); //Obtenemos la normal al triángulo definido por
3 enteros (los 3 j-vértices) del i-triángulo.
//                  fuera = InOut(v,n);
//            }
//      }
//}

            for (int i = 0; i < NumElemsSup && fuera != 1; i++) { //Recorre to-
dos los triángulos de la superficie háptica.
                  for (int j = 0; j < 3; j++){            //Recorre los vérti-
ces de dicho triángulo
                        if (ConnectSup[i][j]-1 == s){     //Si uno de los vér-
tices es el nodo...
                              Vector3<> n;
                              n   =     beam.GetNormal(ConnectSup[i][0],  Con-
nectSup[i][1], ConnectSup[i][2]); //Obtenemos la normal al triángulo definido por
3 enteros (los 3 j-vértices) del i-triángulo.
                              fuera = InOut(v,n);
                        }
                  }
            }

            if (!fuera) fuera = -1;

            /*     La distancia del nodo del Level Set al nodo del Sólido será la
magnitud del vector que une los puntos,
            multiplicada por el negativo del sentido (recordemos que si está
dentro, queremos distancia negativa,
            pero según hemos definido, "sentido" sería 1 en tal caso). */

            mesh->SetDistance(d, fuera * v.GetModulus());
            #endif

            #if INSIDE_OUTSIDE == 0
            mesh->SetDistance(d, v.GetModulus());
            #endif
      }
}

void WriteFile(LSMesh mesh){

      ofstream file;
      file.open ("LSdata.h");

      if (file.is_open()){

            /*     The file will start with the preprocessor data  */
            file << "#pragma once";
            file << endl;

            /*     Now we write down the data */
            file << "//General parameters" << endl;
            file << "const int Nx = " << _Nx << ";" << endl;
            file << "const int Ny = " << _Ny << ";" << endl;
            file << "const int Nz = " << _Nz << ";" << endl;
            file << "const double Dx = " << _Dx << ";" << endl;
            file << "const double Dy = " << _Dy << ";" << endl;
            file << "const double Dz = " << _Dz << ";" << endl;
            file << endl;
            //We save The Vertices
            file << "//Geometric data" << endl;
```

```
		file << "const double LSVertices[" << mesh.GetDimension() << "][" <<
3 << "] = {"; //Ya veremos si es double o Vector3<>

		for (int i = 0; i < mesh.GetDimension(); i++) {
			file << "{";
			Vector3<> teemo = mesh.TCoor2WCoor(mesh.N2Coor(i)) + transla-
tion;
			for (int j = 0; j < 3; j++) {
				file << teemo[j];
				if (j != 2) file << ", ";
			}
			file << "}";
			if (i != mesh.GetDimension()-1) file << ", ";
		}
		file << "};";

		file << endl;

		//We save The Distances
		file << "//Distance data" << endl;
		file << "const double LSMatrix["  << mesh.GetDimension() << "] = {";

		for (int d = 0; d < mesh.GetDimension(); d++) {
			if ((mesh.TCoor2WCoor(mesh.N2Coor(d))[2] + translation[2])<0)
file << -1*mesh.GetDistance(d); //Esto es una prueba rara.
			else file << mesh.GetDistance(d);

			if (d < mesh.GetDimension() - 1) file << ", ";
		}

		file <<"};";

		/*	Save & Close the file	*/
		file.close();
	}
}

int InOut (Vector3<> v, Vector3<> u) {
	double aux = v*u;
	if (aux <= 0) {		//DENTRO
		return 0;
	} else {		//FUERA
		return 1;
	}
}

void GetData()
{

	cout << "Introduzca Delta X (separación en el eje X entre dos puntos del
levelSet consecutivos):" << endl;
	cin >> _Dx;
	cout << "Introduzca Delta Y (separación en el eje Y entre dos puntos del
levelSet consecutivos):" << endl;
	cin >> _Dy;
	cout << "Introduzca Delta Z (separación en el eje Z entre dos puntos del
levelSet consecutivos):" << endl;
	cin >> _Dz;

	cout << "Introduzca Nx (número de puntos que tendrá el levelSet a lo largo
del eje X):" << endl;
	cin >> _Nx;
```

```
	cout << "Introduzca Ny (número de puntos que tendrá el levelSet a lo largo
del eje Y):" << endl;
	cin >> _Ny;
	cout << "Introduzca Nz (número de puntos que tendrá el levelSet a lo largo
del eje Z):" << endl;
	cin >> _Nz;

	cout << "Introduzca la traslación respecto al objeto:" << endl;
	cin >> translation[0];
	cin >> translation[1];
	cin >> translation[2];
}
```

ANEXO D

HARDWARE Y SOFTWARE EMPLEADO

1 DISPOSITIVOS HÁPTICOS

La háptica (del griego *haptikos*: "perteneciente al tacto") designa a la ciencia del tacto, por analogía con la acústica (oído) y la óptica (vista). Por tanto, la tecnología háptica será aquella que esté dedicada a la construcción de instrumentos capaces de percibir y recrear sensaciones táctiles.

Para permitir una respuesta háptica, un dispositivo háptico debe tener algún actuador que aplique fuerzas sobre la piel, y controladores. El actuador proporciona movimiento mecánico en respuesta a un estímulo eléctrico.

Su diseño se puede agrupar por generaciones:

GENERACIÓN I. La mayoría de los diseños tempranos de respuesta háptica usaban tecnologías electromagnéticas tales como motores vibratorios. Estos motores electromagnéticos típicamente operan en resonancia y proporcionan una respuesta háptica fuerte, pero producen una gama limitada de sensaciones y normalmente vibra todo el dispositivo, en lugar de una sección individual. La alerta de vibración en un teléfono móvil es un ejemplo de este tipo de actuadores.

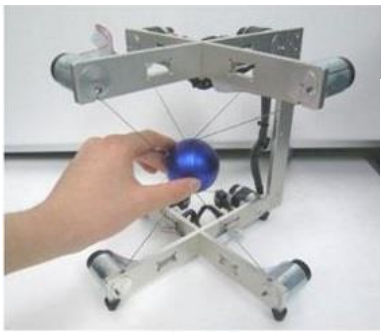
GENERACIÓN II. Estos mecanismos se caracterizan por un mejor control de la respuesta, permitiendo localizar los efectos hápticos en una posición sobre un panel de pantalla táctil, en lugar de todo el dispositivo. Los actuadores de segunda generación incluyen polímeros electroactivos, piezoeléctrico, electrostática y estimulación de la superficie mediante ondas subsónicas. Estos actuadores permiten no sólo alertar al usuario como los dispositivos hápticos primera generación, sino que mejoran la interfaz de usuario con una mayor variedad de efectos hápticos en términos de rango de frecuencia, tiempo de respuesta y la intensidad. Un actuador típico de primera generación tiene un tiempo de respuesta de 35-60ms, un actuador de segunda generación tiene un tiempo de respuesta de 5-15ms.

GENERACIÓN III. Los dispositivos de esta generación proporcionan tanto respuestas específicas dependientes de la coordenada como efectos táctiles personalizables. Los efectos personalizables se crean utilizando chips de control de baja latencia.

GENERACIÓN IV. Tecnología en desarrollo, no disponible comercialmente todavía. Permitiría tener sensibilidad de presión, esto es, que la respuesta cambia según la presión que el usuario ejerza en la interfaz.

El dispositivo del cual hace uso la aplicación aquí desarrollada pertenece a la generación III, ya que la respuesta es personalizable (de lo contrario no se podría hacer la simulación a un nivel háptico) y depende de las coordenadas del punto.

A continuación pueden verse diversos tipos de dispositivos de tercera generación:



El diseño, como puede apreciarse, es muy variado. En el caso del PHANTOM OMNI usado para el proyecto, los actuadores son servomotores colocados en las articulaciones del brazo (incluyendo la base), capaces de dar hasta 3,3 N de fuerza en cada eje principal.

2 OPENGL

OpenGL es una librería multiplataforma con implementación en varios lenguajes de programación para el renderizado de gráficos 2D y 3D.

La API cuenta con numerosas funciones que pueden ser llamadas por el programa cliente, junto con un buen número de constantes predefinidas. Aunque las definiciones de las funciones son similares a C, son independientes del lenguaje de programación, existiendo muchas implementaciones en lenguajes como JavaScript, Java, C y C++.

Al ser multiplataforma, la distribución de una aplicación escrita con OpenGL no tendrá barreras de sistema operativo (por ejemplo, el equivalente de MicroSoft, Direct3D, sólo funciona en Windows).

3 OPENHAPTICS

OpenHaptics es una librería desarrollada por Sensable para el control de dispositivos hápticos. Entre las herramientas que ofrece se incluyen la micro API QuickHaptics, la API del dispositivo háptico (HDAPI), la API de la librería háptica (HLAPI), controladores para el modelo de dispositivo, ejemplos de código, manual del programador y referencia de la API.

Las diferentes API tienen distintas capacidades:

- QuickHaptics está desarrollada para la construcción rápida y sencilla de nuevas aplicaciones hápticas. Su desventaja es que al gestionar por si misma la mayor parte de las acciones del brazo, no deja personalizar los efectos hápticos.
- HLAPI está diseñada para renderizado háptico de alto nivel (en el sentido informático, es decir, alejado de los detalles de funcionamiento de la librería háptica). Su nomenclatura es muy parecida a OpenGL.
- HDAPI permite el acceso de bajo nivel (es decir, código cercano al controlador del hardware, en este caso el dispositivo). Aunque compleja de usar, permite un control muy fino sobre los efectos hápticos, y es la que se utiliza en el proyecto.

4 VISUAL STUDIO 2010

Entorno de desarrollo desarrollado por MicroSoft principalmente para la programación en lenguajes `C`, `C++`, `C#`, VisualBasic y más recientemente, JavaScript y `F#`.

El programa está especialmente diseñado para el desarrollo de aplicaciones para Windows, contando con muchas librerías extra para gestión de ventanas y sistemas propios de Windows.

Tiene un editor de código con IntelliSense (sugiere opciones de auto-completar según el código es escrito por el programador).

También tiene un debugger integrado que trabaja tanto a nivel de código fuente como a nivel de código máquina.

Otras funciones incluyen esquemas de clase, diseño Web, diseñador de clases y esquemas de bases de datos.