Technical Report

IRI-TR-10-10

Simulación de Objetos Planos Deformables con Resolución de Contactos y Colisiones

Santiago Manén Freixa

Institut de Robòtica i Informàtica Industrial

Abstract

El proyecto descrito en esta memoria consiste en la implementación de una aplicación que permita simular la interacción de tejidos entre sí o con objetos rígidos en una plataforma Linux. Se implementa el algoritmo de detección y resolución de contactos y colisiones desarrollado por Bridson, Fedkiw y Anderson (Bridson, Fedkiw, Anderson, 2002). El simulador, *dpoSimulator*, se ha desarrollado para el proyecto de manipulación automática de objetos planos deformables (Proyecto PAU, DPI-2008-06022), el cual tiene como objetivo la manipulación inteligente de tejidos con robots. *dpoSimulator* servirá de base para un futuro desarrollo de un sistema que permita escoger cómo un robot debe manipular un tejido para moverlo de un estado inicial a uno introducido por el usuario.
La memoria consta de cinco apartados. En ellos, se describe en detalle el algoritmo implementado (con los diagramas de flujo apropiados), se provee de un manual de usuario (que muestra los resultados de las simulaciones y explica cómo construir una escena) y un manual de desarrollo (una descripción del flujo principal del programa). La descripción del algoritmo viene acompañada de una considerable cantidad de ecuaciones matemáticas, puesto que se trata de un programa de cálculo. Se demuestran las ecuaciones finales utilizadas y se explican las premisas de partida. También se muestran bastantes figuras que ilustran el funcionamiento del programa o sirven de soporte para explicar los algoritmos implementado. Al final de la memoria, se anexa la documentación de las principales clases del programa, con sus miembros y funciones.
Se adjunta en un CD el simulador desarrollado, *dpoSimulator*, la documentación del mismo y una copia de la memoria.

Institut de Robòtica i Informàtica Industrial (IRI)
Consejo Superior de Investigaciones Científicas (CSIC)
Universitat Politècnica de Catalunya (UPC)
Llorens i Artigas 4-6, 08028, Barcelona, Spain
Tel (fax): +34 93 401 5750 (5751)
http://www.iri.upc.edu

Corresponding author:
S Manén
tel: +34 93 405 4490
smanen@iri.upc.edu
http://www.iri.upc.edu/staff/smanen

PERSONA CIÈNCIA EMPRESA

Universitat Ramon Llull

PROYECTO FINAL DE CARRERA

Título **Simulación de objetos planos deformables con resolución de contactos y colisiones**

Realizado por Santiago Manén Freixa

Dirigido por Guillermo Reyes Pozo

Tutor Juan Andrade-Cetto

Barcelona, 1 de Septiembre 2010

Contenido

1 Introducción

1.1 Objetivo del proyecto

Este proyecto tiene como objetivo el desarrollo de un simulador de objetos planos deformables, tejidos, para un proyecto de investigación sobre la manipulación automática de objetos deformables (Proyecto PAU, DPI-2008-06022). Éste, a su vez, forma parte del proyecto integrado europeo *Perception, Action and Cognition through learning of Object-action Complexes*, PACO-PLUS, fundado por la Comisión Europea. En este apartado, se explica cual es el objetivo, a largo plazo del proyecto de manipulación de objetos deformables y, en el apartado 1.2, se explica cual es el alcance del proyecto que se presenta en este documento.

Se pretende que, dándole a un robot un estado de referencia de una pieza de ropa, en forma de malla, éste decida qué acciones debe ejecutar para llegar a ese estado desde cualquier otro. Es decir, en un comienzo el tejido puede estar en cualquier posición y al robot se le indicará la posición final deseada. Mediante una serie de mediciones, cálculos internos y actuaciones el robot debe manipular la ropa para llegar a la posición final deseada. La principal dificultad de este proyecto radica en la versatilidad que debe tener el sistema desarrollado. Para ello, se plantea el diagrama de bloques general, y simplificado, mostrado cómo Diagrama 1.1.

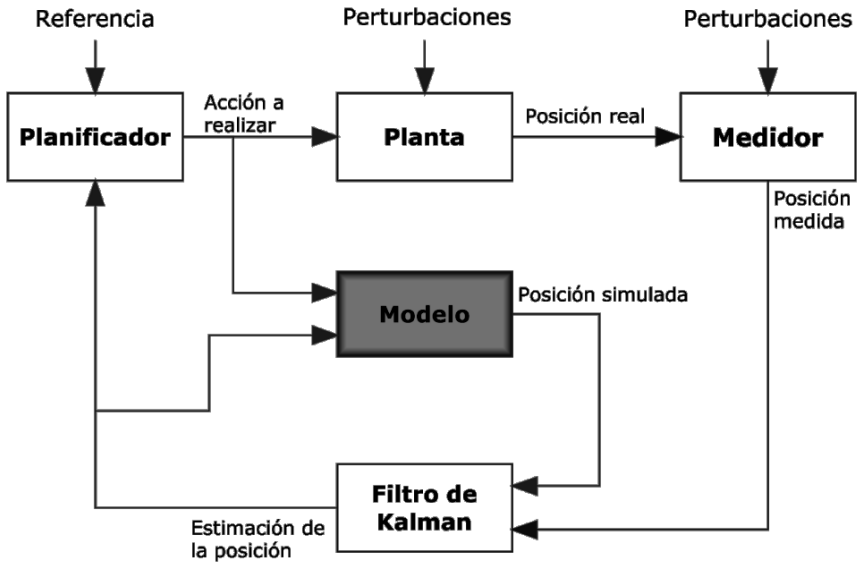


Diagrama 1.1: Diagrama general del proyecto de manipulación automática de objetos deformables. El alcance del proyecto, sombreado en gris, se limita a la simulación de tejidos, según un modelo.

Una malla de referencia, las posiciones futuras deseadas para la ropa, se provee a un planificador. Éste es una aplicación informática que también recibe información sobre el estado actual de la ropa. El planificador decide, en función de estas posiciones y un simulador de tejidos, que acción debe realizar un robot. El robot realiza la acción en un entorno físico, *planta*, que está sujeto a unas perturbaciones, como errores de posicionamiento del manipulador del robot. Fruto de la acción llevada a cabo por el robot, cambia la posición del tejido. Ésta es capturada por un medidor, probablemente una cámara estéreo, que permita determinar la posición de ciertos puntos conocidos de la ropa en un espacio tridimensional. Por supuesto, este medidor no será, ni mucho menos, completamente fiable. Por ello, se simulará, paralelamente mediante un *modelo*, el estado de la ropa al actuar sobre ella el robot. De esta forma, se obtienen dos estados del tejido, uno medido y otro simulado. Cada uno tendrá asignada una covarianza que indicará su fiabilidad, por

lo que se puede proveer esta información, posiciones y covarianzas, a un filtro de Kalman (Andrade-Cetto, 2005) para obtener la mejor estimación posible de la posición de los nodos de la malla del tejido. Esta estimación se proveerá, junto a su covarianza, al planificador cerrando el lazo. El planificador se base en un proceso de decisión parcialmente observable de Markov (POMDP en inglés) el cual decidirá las acciones a realizar en función de su probabilidad de éxito y tendrá que realizar una exploración rápida de los posibles estados alcanzables para encontrar una cadena de acciones óptima (Thrun, Burgard, & Fox, 2006).

La finalización exitosa de este proyecto significará un avance en la robótica, debido a la poca experiencia que se tiene actualmente en la manipulación de objetos deformables. Recientemente, la Universidad de Berkley ha desarrollado un robot que dobla trapos rectangulares, de diferentes medidas y colores, partiendo de cualquier posición inicial, como se muestra en la Figura 1.1. Frente a este robot, el sistema presentado en el Diagrama 1.1 tendrá la ventaja de poder llegar a *cualquier* estado futuro, siempre y cuando sea posible, y mediante una cadena de acciones *planeada por el robot.*

Figura 1.1: El robot PR2 de la Universidad de Berkley puede doblar toallas de diferentes tamaños y colores partiendo de una posición inicial desconocida.

1.2 Alcance del proyecto

Obviamente, el desarrollo de cada uno de los bloques mostrados en el Diagrama 1.1 promete requerir una elevada cantidad de tiempo. Por restricción de tiempo, el alcance de este proyecto es el de desarrollar el simulador resaltado en dicho diagrama.

Es necesario el desarrollo de un simulador personalizado para que se adapte a las necesidades del proyecto de investigación. Inicialmente se probó de adaptar el código de simulación de tejidos del programa de diseño gráfico, de código libre, Blender. Sin embargo, tal intento fue infructuoso debido a la falta de documentación combinada con la rigidez y complejidad del programa, aunque por sí solo da buenos resultados. Sin embargo, se ha encontrado una librería que sí es lo suficientemente flexible para las necesidades del proyecto. *Freecloth* es una librería de código libre que simula el comportamiento de tejidos basándose únicamente en la dinámica interna (Pritchard, freecloth project, 2003). Esto quiere decir que no puede simular colisiones entre objetos, ni de un objeto consigo mismo. Ésta es una gran limitación que impide la simulación de la mayoría de situaciones en las que el proyecto resultaría útil. Ante esta contrariedad, la principal aportación de este Proyecto Final de Carrera es el desarrollo de un simulador de tejidos que tenga en cuenta los contactos y colisiones con otros objetos. Esta aplicación debe ser versátil, permitiendo la simulación de cualquier escena y con cualquier tipo de manipulación de objetos, y fácilmente ampliable, pues en un futuro será esclava del planificador.

1.3 Contenido de la memoria

El contenido principal de la memoria se encuentra en el apartado de Desarrollo. Éste tiene tres partes:

- **2.1 Trabajos previos:** En esta sección se indica cual es el estado del arte y en orden cronológico, las aportaciones de los principales autores al campo de la simulación de tejidos. También se indica qué algoritmo se ha implementado y se justifica su elección.

- **2.2 Algoritmo implementado:** Este apartado detalla el algoritmo que se implementa en el simulador desarrollado, *dpoSimulator*. Para ello, primero se describe de forma general e intuitiva el algoritmo, para después entrar en más detalle y con las demostraciones matemáticas necesarias para llegar a las expresiones finales de cálculo.

- **2.3 Descripción del simulador desarrollado:** Este apartado consta de una guía de usuario del programa, que explica cómo crear, cargar y simular escenas. También incluye una guía para el desarrollador, explicando cuales son las partes más importantes del programa, las clases de la capa exterior y dando recomendaciones para el desarrollador.

El tercer apartado, Conclusiones, lista los objetivos cumplidos e incluye también un apartado de Recomendaciones, que incluye sugerencias sobre cómo ampliar el programa.

Finalmente, los anexos incluyen la documentación, en inglés, de las principales clases del simulador. Éstas son referenciadas en el apartado 2.2, en qué clase se implementa cada algoritmo.

2 Desarrollo

2.1 Trabajos previos

En esta década, la simulación de tejidos ha cobrado un gran interés, en el mundo de la animación, facilitado por la creciente potencia de cálculo de los ordenadores. Por ello, hay mucha investigación documentada sobre este tema y muchos algoritmos que priorizan diferentes aspectos de la simulación.

En 1995, un artículo escrito por Xavier Provot revolucionó la simulación de tejidos (Provot, 1995). Provot planteaba un modelo de simulación de la dinámica interna de los tejidos basado en redes de muelles. Es decir, tras mallar el tejido, los nodos se unían con tres tipos de muelles: de tracción (que unían nodos adyacentes), de cizalla (que unían nodos en diagonal) y de flexíon (que unían dos nodos saltándose el del medio). Provot planteaba el equilibrio de fuerzas para todos los nodos y simulaba mediante una integración explícita. Puesto que las constantes de tracción de la ropa son muy elevadas, debido a que la ropa no se puede estirar mucho sin desgarrarla, la integración explícita requería pasos muy pequeños. Además, la concentración de tensiones en algunos nodos, por ejemplo en el caso de un trapo colgando de dos esquinas, provocaba una gran deformación elástica, que se veía muy artificial. Provot solucionó esta *hiperelasticidad* limitando la deformación máxima de los muelles. Esto tiene su fundamento físico, pues aproxima al comportamiento del material en la zona plástica. El modelo de Provot, frente a sus carencias, es uno de los más utilizados debido a su sencillez.

Baraff y Witkin plantearon un algoritmo de simulación de la dinámica interna de los tejidos basándose en una integración implícita de las ecuaciones de balance fuerzas (Baraff & Witkin, 1998). La integración implícita permite avanzar la simulación a grandes pasos, con incrementos de tiempos grandes, sin que se vea afectada la estabilidad del sistema, a diferencia de la integración explícita de Provot. Esto se debe a que la estabilidad de la simulación con una integración implícita se mantiene incluso si se simulan pasos grandes. Sin embargo, este tipo de simulación requiere la resolución de un sistema disperso de elevadas dimensiones. Por ello, y aprovechando el hecho de que la matriz de coeficientes es simétrica, se resuelve el sistema mediante el método del Gradiente Conjugado. Además, Baraff plantea un modelo energético de la ropa. Es decir, en vez de basarse en las fuerzas debidas a muelles entre los nodos, se plantean las fuerzas como derivadas de la energía potencial de la ropa. Se ha utilizado este algoritmo, para simular la dinámica interna de los tejidos, debido a su rapidez y disponibilidad en la librería de código libre *Freecloth* (Pritchard, 2009).

Volino razona que la diferente naturaleza de las fuerzas internas entre los nodos de la ropa frente a las fuerzas puntuales debidas a contactos, requiere dos métodos de simulación diferentes. Se separa la simulación de la dinámica interna de los tejidos (tracción, cizalla y flexión) de la simulación de contactos y colisiones (Volino & Thalmann, 2000). Además, también desarrolla un algoritmo de simulación, priorizando la velocidad de simulación respecto la calidad de la misma. Su algoritmo se basa en el de Baraff y Witkin (Baraff & Witkin, 1998) , con la particularidad de que la integración se realiza en el punto medio, en vez del punto inicial. Esto elimina un amortiguamiento artificial que se aprecia en la simulación de tejidos con pasos grandes (Volino & Thalmann, 2000). Asimismo, Volino desarrolla un algoritmo de colisiones, particular para su modelo de la dinámica interna, basado en la aplicación de restricciones de posición y velocidad sobre las partículas que van a colisionar. Se ha decidido no implementar este algoritmo de colisiones, pues no es independiente del modelo de la dinámica interna.

En 2002, Bridson, Fedkiw y Anderson publicaron un artículo que revolucionó la simulación de interacciones entre tejidos y objetos rígidos (Bridson, Fedkiw, & Anderson, 2002). Éste estaba dedicado completamente a un algoritmo que se podía aplicar sobre cualquier simulador de

dinámica interna de tejidos. Plantearon un tratamiento híbrido de dos tipos de interacciones entre objetos:

- **Contactos:** Los contactos ocurrían debido a la proximidad de dos objetos al principio de un paso. Éstos se resolvían aplicando impulsos de repulsión que separaban los dos objetos.
- **Colisiones:** Las colisiones ocurrían durante un paso simulado y se resolvían aplicando impulsos a los objetos, garantizando una velocidad relativa nula de los dos objetos en su dirección normal, pues la colisión se considera inelástica.

Este algoritmo de simulación de tejidos con colisiones considera la simulación de dinámica interna como una *estimación*, en la que detecta y resuelve contactos y colisiones. Éste es el algoritmo implementado en el simulador desarrollado en este proyecto, *dpoSimulator*, debido a la calidad de la simulación, a que se puede aplicar sobre el algoritmo de Baraff y a la claridad y precisión con la que está documentado. El algoritmo completo se describe en el apartado 2.2.

Baraff, Witkin y Kass desarrollaron, el 2003, un algoritmo que trataba un caso concreto de colisión que solía provocar problemas, el pellizco, *pinch* (Baraff, Witkin & Kass, 2003). Éste fenómeno ocurría, por ejemplo, en la simulación de una camisa en la zona del codo. La ropa quedaba *pellizcada*, atravesándose forzosamente, y lo seguía durante toda la simulación. Baraff, Witkin y Kass plantean un algoritmo para resolver este caso concreto.

Por último, cabe comentar que Provot, en el 2004, desarrolló un algoritmo de simulación de tejidos con colisiones (Provot, 2004) para su modelo de dinámica interna (Provot, 1995). Éste no se ha implementado, pues habría requerido el desarrollo adicional del simulador de la dinámica interna de Provot.

2.2 Algoritmo implementado

A continuación se describe en profundidad el algoritmo implementado. Se empieza por una descripción general e intuitiva, apartado 2.2.1, que describe la estrategia general del algoritmo. Después se introducen los tipos de elementos geométricos y objetos que se utilizarán más adelante, apartado 2.2.2. Y finalmente, se describen los algoritmos de detección y resolución de contactos y colisiones y se demuestran las ecuaciones matemáticas que se utilizan en el cálculo, apartados 2.2.3 a 2.2.5.

2.2.1 Descripción general del algoritmo de simulación con contactos y colisiones

El algoritmo que se ha implementado en este proyecto, para desarrollar el simulador *dpoSimulator*, tiene la función de detectar y resolver los contactos y colisiones que ocurren entre objetos planos deformables, tejidos, consigo mismos u otros objetos, ya sean rígidos o deformables. Estos contactos y colisiones aparecen al simular basándose únicamente en la *dinamica interna* de los objetos. Ésta tiene en cuenta:

- Los esfuerzos de tracción, flexión y cizalla que se deben a la posición relativa de los nodos de la malla del tejido.
- Fuerzas externas que actúan durante un periodo largo de tiempo. Éstas son comúnmente la gravedad y la fuerza del viento.
- Restricciones de posición y velocidad de los nodos. Éstas permiten manipular los tejidos. Por ejemplo, para levantar una pieza por un nodo simplemente se le tiene que restringir la velocidad de ese nodo para que su tercera componente sea positiva.

- Fricciones internas del tejido, debido a la velocidad relativa de los nodos que componen su malla.

Se utiliza la librería de código libre *freecloth* (Pritchard, freecloth project, 2003) para simular la dinámica interna del tejido. Ésta permite avanzar la simulación de un objeto plano deformable, con un incremento de tiempo determinado por el usuario, implementando el algoritmo descrito en (Baraff & Witkin, Large Steps in Cloth Simulation, 1998). El cual calcula las fuerzas que actúan sobre los nodos del tejido, derivando la energía potencial, y las introduce en un sistema de ecuaciones que permite calcular la velocidad y la posición de los mismos, ecuación (2.1).

$$\frac{d}{dt}\begin{pmatrix}\mathbf{x}\\ \mathbf{v}\end{pmatrix} = \begin{pmatrix}\mathbf{v}\\ \mathbf{M^{-1}f(x,v)}\end{pmatrix} \tag{2.1}$$

Dónde:

x, **v:** Vector columna de posición y velocidad de todos los nodos del tejido. Suponiendo que la malla tenga *n* nodos, estos vectores serán de *3n* por 1 dimensiónes.

f: Vector que contiene las fuerzas que actúan sobre los nodos, en función de la posición de éstos.

M: Matriz de masa. Es una matriz diagonal en la que el elemento m_{ii} es la masa del nodo *i* del tejido.

La ecuación anterior se puede discretizar, sustituyendo las derivadas por incrementos. Haciendo una aproximación de Taylor, de primer orden, de la fuerza, tomando como referencia las fuerzas que actúan en el tiempo actual *n*, y aislando el termino de incremento de velocidad en el paso simulado, se obtiene el sistema (2.1). Éste se resuelve por el método del Gradiente Conjugado modificado (Baraff & Witkin, Large Steps in Cloth Simulation, 1998), obteniendo el incremento de velocidades que se produce en el paso simulado. A partir del incremento de velocidades, se puede calcular la variación de posiciones con la ecuación (2.3).

$$\left(\mathbf{I} - h\mathbf{M^{-1}}\frac{\mathbf{df}}{\mathbf{dv}} - h^2\mathbf{M^{-1}}\frac{\mathbf{df}}{\mathbf{dx}}\right)\Delta\mathbf{v} = h\mathbf{M^{-1}}\left(\mathbf{f_0} + h\frac{\mathbf{df}}{\mathbf{dx}}\mathbf{v_0}\right) \tag{2.2}$$

$$\Delta\mathbf{x} = (\Delta\mathbf{v} + \mathbf{v_0})h \tag{2.3}$$

Dónde:

h: Duración del paso simulado.

$\frac{\mathbf{df}}{\mathbf{dv}}$, $\frac{\mathbf{df}}{\mathbf{dx}}$: Jacobianos de la fuerza respecto a la velocidad y la posición al inicio del paso simulado.

$\mathbf{v_0}, \mathbf{f_0}$: Velocidad de los nodos y fuerzas que actúan sobre ellos al inicio del paso simulado.

$\Delta\mathbf{x}$, $\Delta\mathbf{v}$: Variación de posición y de velocidad que sufren los nodos durante el paso simulado.

Se observa, en la ecuación (2.3), que se ha utilizado una formulación implícita, pues se cumple la ecuación (2.1) si la velocidad es la que existe al final del paso. Por ello, se pueden simular pasos grandes sin problemas de estabilidad. No se entra más en detalle en el algoritmo de Baraff y Witkin pues no se ha implementado, sino aprovechado de la librería *freecloth*. Para más información sobre este algoritmo, se refiere al lector al documento (Baraff & Witkin, Large Steps in Cloth Simulation, 1998).

La simulación de la dinámica interna no tiene en cuenta las interacciones, colisiones, entre el tejido y otros objetos, o incluso consigo mismo. Por ello, se deben detectar estas interacciones y se deben resolver de la forma más realista posible. Para ello, se ha decidido implementar el algoritmo descrito en (Bridson, Fedkiw, & Anderson, 2002), como se justifica en el apartado 2.1. Éste plantea la siguiente estrategia:

1. Se define un incremento de tiempo Δt desde el tiempo actual t_n al siguiente t_{n+1}.
2. Se simula un paso, de duración Δt, teniendo en cuenta únicamente la dinámica interna. Esto se puede hacer con cualquier simulador de dinámica interna, en este caso el de la librería *freecloth*. De esta forma, se obtienen unas posiciones estimadas $\tilde{\mathbf{x}}^{n+1}$ y unas velocidades estimadas $\tilde{\mathbf{v}}^{n+1}$ para el final del paso.
3. Se calculan las velocidades medias estimadas de los nodos durante el paso con la ecuación (2.4).

$$\tilde{\mathbf{v}}^{\mathbf{n+1/2}} = \frac{\tilde{\mathbf{x}}^{\mathbf{n+1}} - \tilde{\mathbf{x}}^{\mathbf{n}}}{\Delta t} \tag{2.4}$$

4. Se detectan los contactos que ocurren en el tiempo t_n y se resuelven aplicando impulsos de repulsión a las velocidades medias estimadas para obtener una nueva estimación. El apartado 2.2.3 explica como se detectan los contactos y el 2.2.5 como se resuelven.
5. Se detectan las colisiones que ocurren en el periodo simulado, con las velocidades medias estimadas, y se resuelven aplicando impulsos a las velocidades medias estimadas. Es posible que se sigan produciendo colisiones, debido a que los impulsos aplicados sobre cada nodo se promedian, por lo que se vuelven a detectar y resolver colisiones de forma iterativa, hasta que se han solucionado todas las colisiones. En el apartado 2.2.4, se explica cómo se detectan las colisiones, y en el 2.2.5, se explica cómo se resuelven.
6. En este punto, las velocidades medias ya son definitivas, pues llevan a una configuración libre de colisiones. Se calculan las posiciones al final del paso a partir de las velocidades medias modificadas, ecuación (2.5).

$$\mathbf{x}^{\mathbf{n+1}} = \mathbf{x}^{\mathbf{n}} + \Delta t \mathbf{v}^{\mathbf{n+1/2}} \tag{2.5}$$

7. Finalmente, se actualizan las velocidades al final del paso, $\mathbf{v}^{\mathbf{n+1}}$, para que concuerden con las velocidades medias modificadas. La actualización de velocidades se explica en el apartado 2.2.5.

Se repiten estos pasos hasta que se haya cubierto todo el intervalo que se quiere simular. Los apartados que siguen describen en detalles cómo se ejecuta cada uno de estos pasos. El Diagrama 2.1 muestra de forma gráfica el algoritmo descrito.

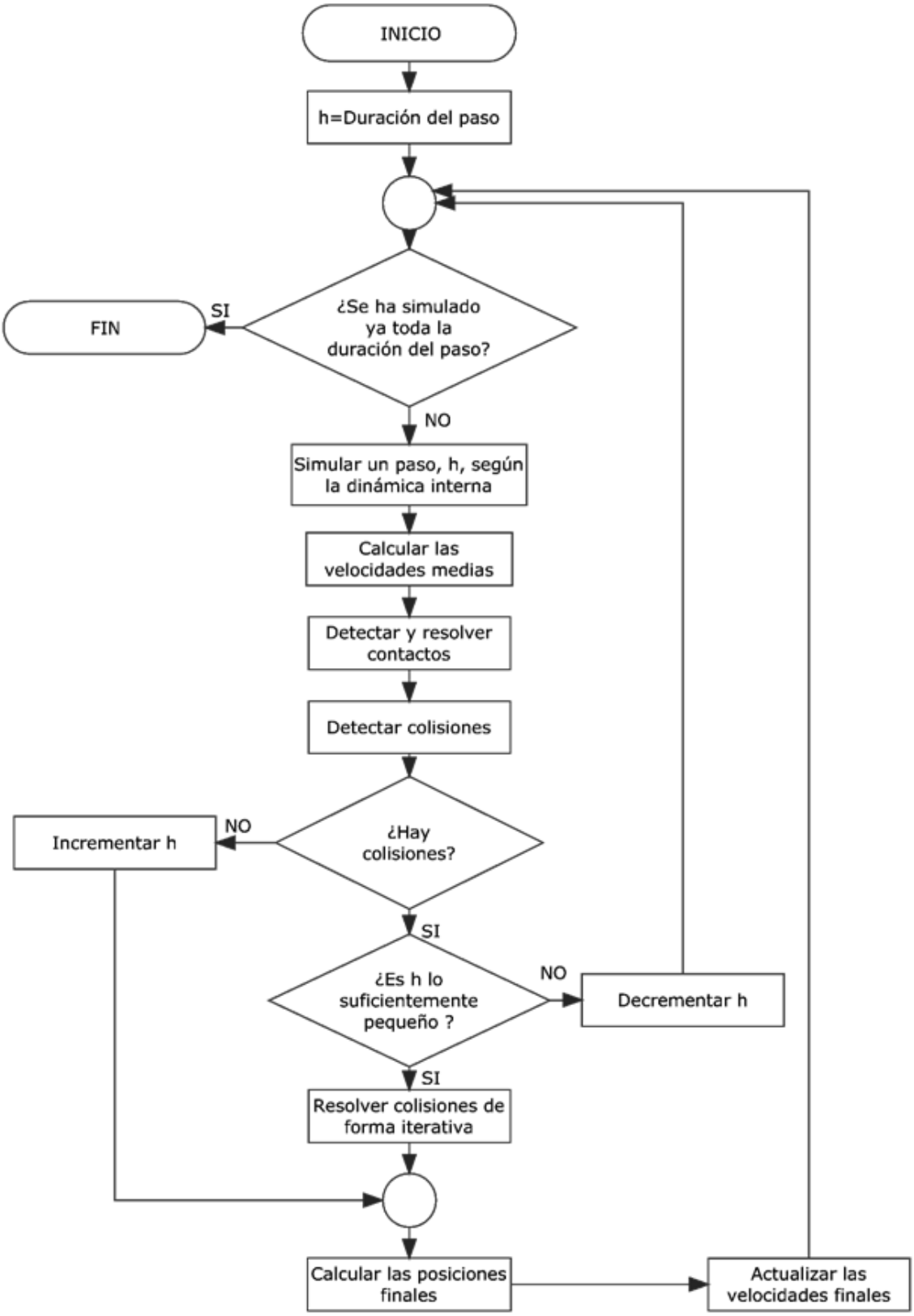


Diagrama 2.1: Algoritmo general de simulación de tejidos con resolución de contactos y colisiones. El algoritmo se corresponde a la simulación de un solo paso.

2.2.2 Modelado geométrico

La simulación se basa en el método de los elementos finitos, MEF, describiendo la geometría de los objetos físicos en mallas *triangulares*. A continuación, se introducen los elementos geométricos, llamados *entidades* geométricas, que cobran importancia en el algoritmo en el que se basa el simulador (Bridson, Fedkiw, & Anderson, 2002).

Entidades geométricas

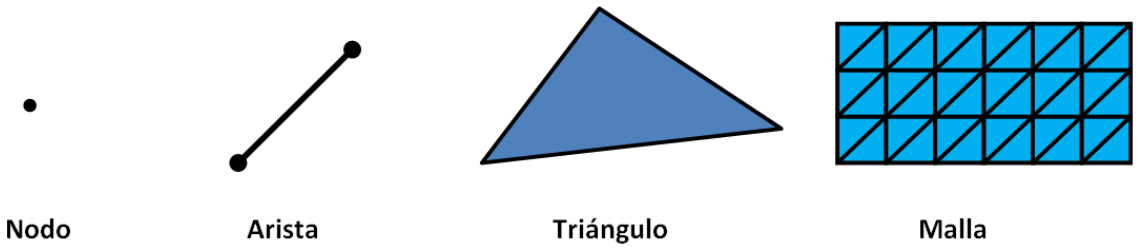


Figura 2.1: Tipos de entidades geométricas contempladas.

Es necesario destacar cuales son las entidades geométricas utilizadas y qué función tienen en la simulación. En la Figura 2.1 se muestran las entidades geométricas, siendo éstas:

1. **Nodo:** El nodo es la unidad más pequeña de la malla. Cada nodo de un objeto tiene asociado información sobre su posición, su velocidad y su aceleración, que es resultado de las fuerzas que actúan sobre él. Por ello, cualquier cálculo que requiera información, de posiciones y velocidades, la obtendrá de los nodos. Y cualquier transformación que se realice sobre el objeto, también se realizará sobre los nodos. Por ejemplo, los impulsos debidos a contactos y colisiones se calculan como resultantes sobre aristas y triángulos pero se deben repartir a los nodos, para más información consulte el apartado 2.2.5. El nodo una de las entidades que puede entrar en contacto o colisionar con otras entidades del mismo u otro objeto.
2. **Arista:** La arista es una entidad *ficticia* que está formada por dos nodos. Se crea esta entidad por dos razones:
   - Para representar gráficamente la malla.
   - Para detectar contactos y colisiones entre aristas, apartado 2.2.3 y 2.2.4.

   La posición de un punto que pertenece a una arista se puede calcular en función de las posiciones de los vértices de las aristas y de un parámetro *a*, parecido a las coordenadas baricéntricas, ver la ecuación (2.6). Este parámetro deberá estar entre 0 y 1. La velocidad del punto, en función de las velocidades de los vértices de la arista, se puede calcular derivando la ecuación de la posición.

$$\mathbf{x} = a\mathbf{x_1} + (1-a)\mathbf{x_2} \tag{2.6}$$

$$\mathbf{v} = a\mathbf{v_1} + (1-a)\mathbf{v_2} \tag{2.7}$$

Dónde:

$\mathbf{x}, \mathbf{v}$**:** Posición y velocidad de un punto que pertenece a una arista.

$\mathbf{x_1}, \mathbf{v_1}$**:** Posición y velocidad del vértice 1 de la arista 1.

$\mathbf{x_2}, \mathbf{v_2}$**:** Posición y velocidad del vértice 2 de la arista 2.

a: Distancia desde el punto hasta el vértice 1 respecto a la longitud total de la arista, en tanto por uno.

3. **Triángulo:** Esta entidad geométrica se utiliza para la detección de contactos y colisiones de forma aproximada y de forma precisa, contactos y colisiones nodo-triángulo. La posición de un punto en un triángulo se puede calcular con sus *coordenadas baricéntricas*. Éstas son las que hacen que se cumplan las ecuaciones (2.8) y (2.9), dónde w_1, w_2 y w_3 son las coordenadas baricéntricas del punto respecto al triángulo. El punto se encuentra dentro del área del triángulo si y sólo si sus coordenadas baricéntricas están entre 0 y 1.

$$\mathbf{x} = w_1\mathbf{x_1} + w_2\mathbf{x_2} + w_3\mathbf{x_3} \tag{2.8}$$

$$\mathbf{v} = w_1\mathbf{v_1} + w_2\mathbf{v_2} + w_3\mathbf{v_3} \tag{2.9}$$

4. **Malla:** La malla de un objeto, sólo se considera como una unidad en la simulación de la dinámica interna (Baraff & Witkin, Large Steps in Cloth Simulation, 1998) y la actualización de velocidades, para más información consulte el apartado 2.2.5.

Tipos de objeto

El simulador desarrollado, *dpoSimulator*, distingue dos tipos de objeto:

- **Objetos planos deformables:** Abreviados como *dpo* (del inglés Deformable Planar Objects), éstos se modelan como mallas planas que se pueden deformar, representando el comportamiento de tejidos reales ante influencias externas. Los tejidos tienen un grosor determinado, por lo que la malla representa su *fibra neutra*, como se muestra en la Figura 2.29 en la página 48. La calidad de la simulación depende directamente de los incrementos de tiempo y de la densidad de la malla. La simulación de tejidos es la función principal del simulador y consta de dos etapas principales:
    - **Simulación de la dinámica interna:** La simulación de la dinámica interna tiene en cuenta la tracción, cizalla y flexión de las fibras, la gravedad y fricciones internas. Sin embargo, no tiene en cuenta las colisiones que se pueden producir entre el tejido consigo mismo o con otros objetos. La dinámica interna no se ha desarrollado para este proyecto, sino que se ha aprovechado de la librería de código libre *freecloth*. Ésta, como se explica en el apartado 2.2.1, implementa el algoritmo descrito en (Baraff & Witkin, Large Steps in Cloth Simulation, 1998).
    - **Resolución de contactos y colisiones:** Tomando el resultado de la dinámica interna como primera aproximación, se resuelven los contactos y colisiones que aparecen, según el algoritmo descrito en (Bridson, Fedkiw, & Anderson, 2002). La implementación de dicho algoritmo es la principal aportación de este proyecto.

- **Objetos rígidos:** Los objetos rígidos no se pueden deformar. El algoritmo de resolución de contactos y colisiones sólo sirve para interacciones en las que interviene, por lo menos, un objeto deformable. Por ello, el simulador *dpoSimulator* supone que todos los objetos rígidos tienen una masa infinita, no viéndose afectados por la interacción con otros objetos, como se muestra en la Figura 2.2.

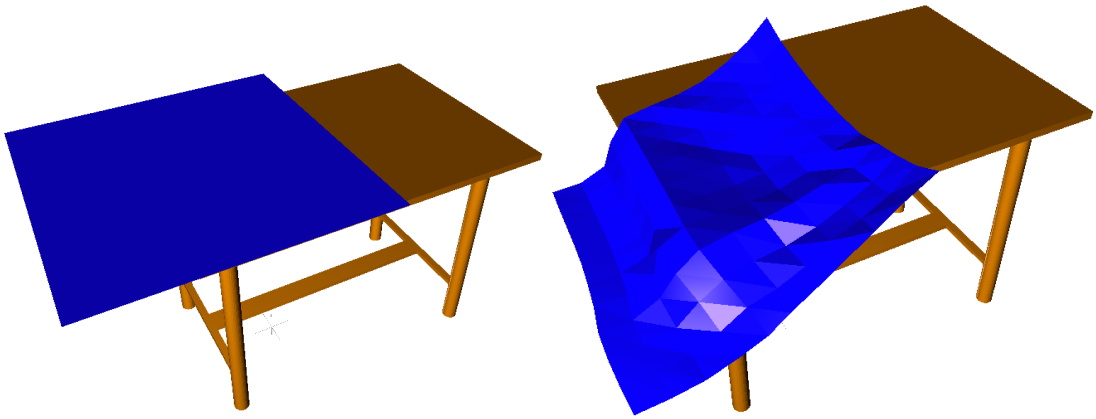


Figura 2.2: En la escena ClothOnTable la mesa es un objeto rígido, por lo que ni se mueve ni se deforma al interaccionar con el tejido.

2.2.3 Detección de contactos

Según (Bridson, Fedkiw, & Anderson, 2002) se produce un *contacto* cuando dos *entidades* están a una *distancia normal* inferior a una *distancia de referencia* determinada y además se cumplen otra serie de requisitos geométricos, detallados en las secciones que siguen. Esta pareja de *entidades* puede ser del tipo nodo-triángulo o arista-arista, pues el algoritmo trabaja con ambos tipos de contacto.

La *distancia de referencia* depende del tipo de objeto al que pertenecen las dos entidades. Físicamente, la distancia de referencia se corresponde con aquella distancia, entre los dos puntos más cercanos de las dos entidades, por debajo de la cual empiezan a producirse fuerzas repulsivas, debido a la compresión del tejido. Esto se puede traducir a una distancia mínima, llamada *distancia de referencia*, por debajo de la cual se considera que hay contacto. En la Figura 2.3 se muestra un caso en el que un nodo *i* está en contacto con un triángulo *j*, debido a que se encuentra a una distancia inferior a la de referencia, representada por la línea discontinua. Este mismo concepto se puede extrapolar a los contactos entre aristas.

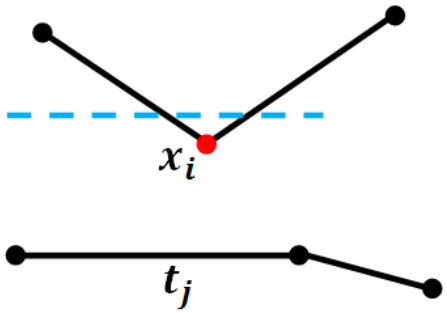


Figura 2.3: El nodo xi está en contacto con el triángulo tj

Las entidades en contacto pueden pertenecer al mismo objeto, un tejido que contacta consigo mismo, o a objetos diferentes, como en el caso de un trapo en reposo sobre una mesa. De esta

forma, cada entidad pertenece a un objeto deformable o a un objeto rígido. La distancia de referencia se calcula según la ecuación (2.99), donde h_1 es la distancia de referencia del objeto al que pertenece la entidad 1 y h_2 es la distancia de referencia del objeto al que pertenece la entidad 2. Este promedio es debido a que las mallas de objetos deformables representan la fibra neutra del objeto real.

$$h_{ref} = \frac{h_1 + h_2}{2} \tag{2.10}$$

La distancia de referencia del objeto se corresponde con su grosor, por lo que es nula para objetos rígidos. Por ejemplo, si se quiere determinar si un nodo de una malla, que representa un pañuelo de seda de 0,5 mm de espesor, está en contacto con un maniquí, objeto rígido, la distancia de referencia será de 0,25 mm. Esto se debe a que la malla de un objeto deformable se toma como *fibra neutra,* mientras que la de un objeto rígido se toma como su capa exterior.

Se podría plantear la detección de contactos entre dos objetos como una comprobación de proximidad geométrica entre cada una de sus entidades, nodo-triángulo y arista-arista. Sin embargo, el número de comprobaciones de proximidad que se debería hacer aumentaría muy rápidamente con respecto al número de nodos, ralentizando mucho la detección de proximidades.

A continuación se presenta un caso sencillo que ilustra lo comentado. Se plantea la detección de contactos entre las entidades presentes en una malla con forma de tira, cómo se puede ver en la Figura 2.4.

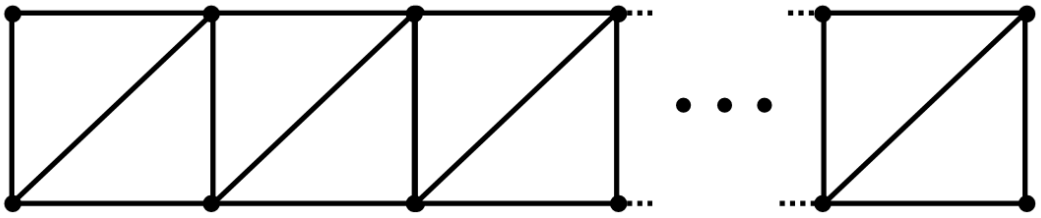


Figura 2.4: Malla con forma de tira

Se plantea el estudio del aumento de comprobaciones de contactos a realizar entre las entidades de dicha malla en función del número de nodos. A partir del número de nodos se puede calcular el resto de parámetros de la malla de forma recursiva como se muestra en las ecuaciones (2.11) y (2.12).

$$a_n = a_{n-1} + 2 \tag{2.11}$$

$$c_n = c_{n-1} + 1 \tag{2.12}$$

Dónde:

a_n: Número de aristas para *n* nodos.

c_n: Número de caras para *n* nodos.

Se parte de los valores iniciales:

$$a_3 = 3 \tag{2.13}$$

$$c_3 = 1 \tag{2.14}$$

También de forma recursiva, se puede calcular el número de comprobaciones de contactos a realizar, para la malla con forma de cinta, mediante las ecuaciones (2.15) y (2.16).

$$d_n^{nt} = d_{n-1}^{nt} + c_{n-1} + n - 3 \tag{2.15}$$

$$d_n^{aa} = d_{n-1}^{aa} + 2 \cdot (a_{n-1} - 1) \tag{2.16}$$

Dónde:

d_n^{nt}: Número de comprobaciones de proximidad entre nodos y triángulos, tipo nodo-triángulo, necesarias.

d_n^{aa}: Número de comprobaciones de proximidad entre aristas, tipo arista-arista, necesarias.

n: Número de nodos que componen la malla.

Se parte de los valores iniciales:

$$d_3^{nt} = 0 \tag{2.17}$$

$$d_3^{aa} = 0 \tag{2.18}$$

A partir de las ecuaciones recursivas (2.15) y (2.16), se puede calcular el número de comprobaciones a realizar en función del número de nodos. Las ecuaciones (2.19) y (2.20) calculan el número de comprobaciones nodo-triángulo y arista-arista a realizar, en función del número de nodos de la malla con forma de tira.

$$d_n^{nt} = n^2 - 5n + 6 \tag{2.19}$$

$$d_n^{aa} = 2n^2 - 10n + 12 \tag{2.20}$$

El Gráfico 2.1 muestra el rápido crecimiento del número de comprobaciones de contactos a realizar, entre las entidades de una malla tipo tira, en función del número de nodos. A partir de las ecuaciones (2.19) y (2.20), se deduce que el orden de cálculo de comprobaciones es cuadrático con respecto al número de nodos. Ésta es una norma general que se cumple para la mayoría de las mallas. Esto significa que, en caso de utilizar esta metodología, al duplicar el número de nodos de una malla el tiempo de cálculo se cuadruplicaría.

Se puede extrapolar fácilmente lo dicho a la detección de contactos entre dos mallas distintas. Suponiendo que una malla tenga *m* nodos y la otra *n*, el número de comprobaciones de contactos entre las entidades de una con respecto a la otra será de orden *m·n.*

En (Bridson, Fedkiw, & Anderson, 2002), se plantea una alternativa que se basa en realizar un filtrado preliminar que simplifique el número de casos a comprobar con precisión posteriormente. Es decir, se plantean dos pasos:

1. **Detección aproximada de contactos:** Consiste en desestimar, mediante cálculos poco costosos, aquellas parejas de entidades que *seguro* que no están en contacto. Mediante

esta detección aproximada de contactos se obtiene una lista de parejas de triángulos que *puede que estén* en contacto.

2. **Detección precisa de contactos:** Se realiza una detección precisa entre las entidades, nodo-triángulo y arista-arista, que forman parte de las parejas de triángulos *preseleccionadas* mediante la detección aproximada anterior.

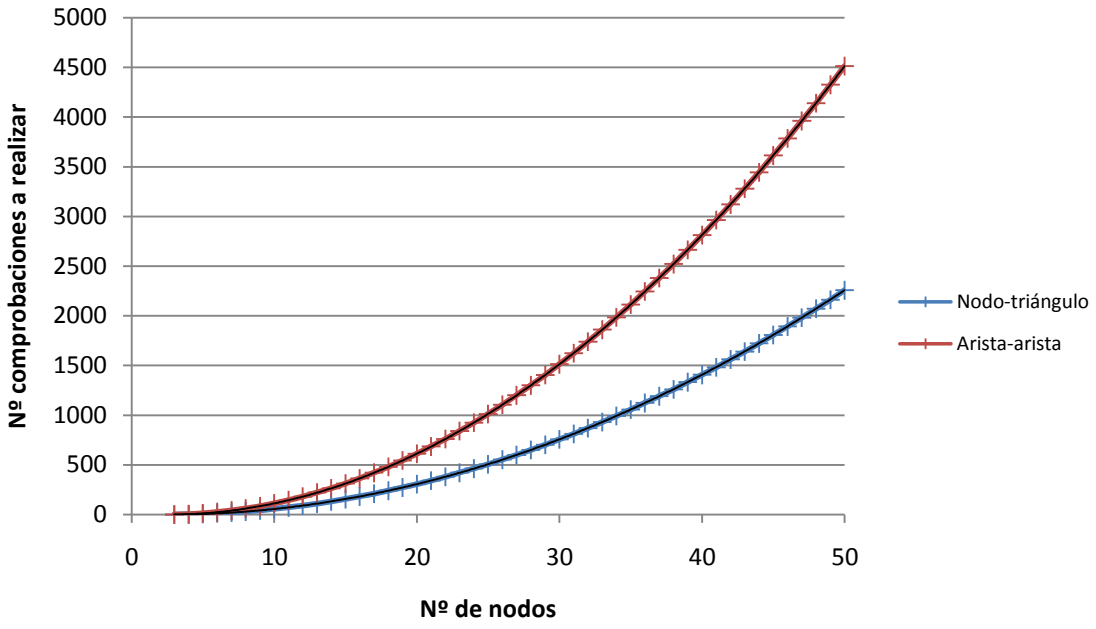


Gráfico 2.1: Relación cuadrática entre el número de nodos y el número de comprobaciones de proximidad a realizar.

A continuación se detallan los algoritmos de detección de contactos de forma aproximada y precisa implementados en la aplicación desarrollada.

Detección aproximada de contactos

La detección aproximada de contactos consiste en una serie de cálculos sencillos que permiten obtener una selección de parejas de triángulos que *es posible* que estén en contacto. Un cálculo posterior más preciso realizado sobre esta preselección permitirá determinar que entidades están en contacto. En los apartados que siguen, se explica cómo se ha utilizado cajas delimitadoras alineadas con los ejes para realizar una detección aproximada de contactos.

Cajas delimitadoras

La caja delimitadora, o *Bounding Box*, de una malla cualquiera es aquella que contiene a toda la malla. Se utilizan ampliamente en el mundo de la simulación, animación y programación gráfica para realizar comprobaciones sencillas. De cara a la detección de contactos y de colisiones, interesa la esta caja sea lo más pequeña posible, para descartar contactos. Como se muestra en la Figura 2.5, las cajas delimitadoras pueden tener distintas orientaciones. En el simulador se implementan cajas delimitadoras alineadas con el sistema de coordenadas (Axis Aligned Bounding Boxes o AABB), en las que todas sus aristas son paralelas a los ejes del sistema de coordenadas global. Otra opción sería alinear la caja delimitadora de una malla de tal manera que ocupase el mínimo tamaño posible, correspondiente al caso *b* en la Figura 2.5. Sin embargo, ello supondría un mayor coste computacional que probablemente no se vería compensado con una mayor rapidez de detección aproximada de contactos y de colisiones.

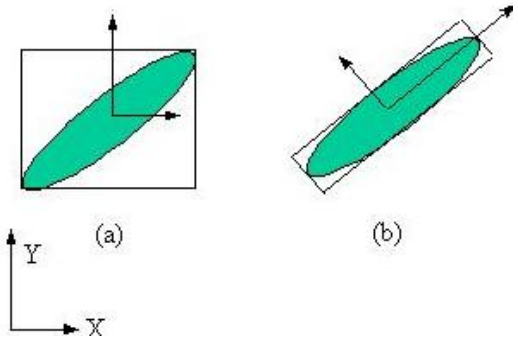


Figura 2.5: (a) Caja delimitadora alineada con ejes (AABB). (b) Caja delimitadora no alineada con ejes (BB)

La caja delimitadora alineada con los ejes, que encierra una malla en el simulador de tejidos, será aquella que contenga todos los nodos de la malla. Por ello, su cálculo se basará simplemente en encontrar el mínimo y máximo de los componentes de posición (x, y, z) de todos los nodos. Por ejemplo, la componente *x* mínima se corresponderá con la componente *x* más pequeña entre todas las de todos los nodos que debe contener la caja delimitadora. Cada caja delimitadora se define con los siguientes parámetros:

- Coordenadas mínimas: *x, y, z mínimas*
- Coordenadas máximas: *x, y, z máximas*

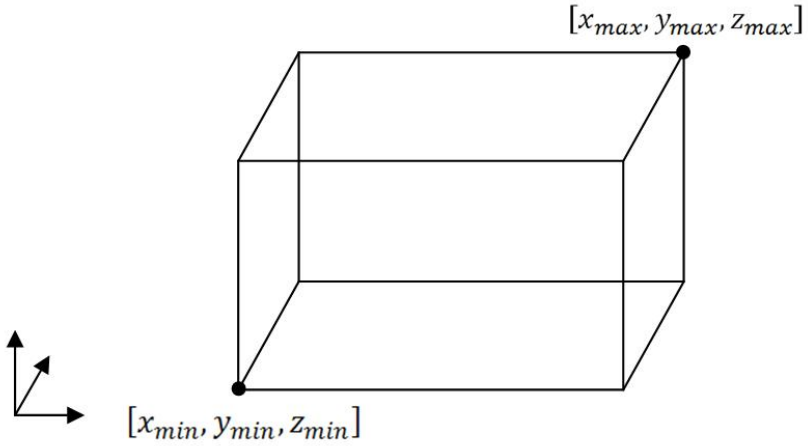


Figura 2.6: Una AABB se define por las coordenadas de sus esquinas

Para la detección de contactos, se deben crear las cajas delimitadoras de forma tal que tengan en cuenta el espesor de los objetos sólidos deformables.

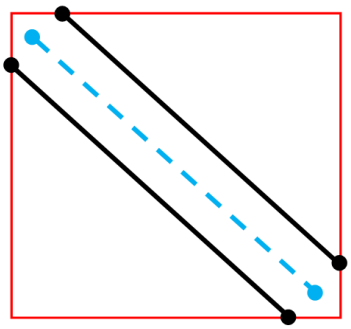


Figura 2.7: La caja delimitadora (roja) para detección de contactos tiene en cuenta el espesor de un triángulo (visto de perfil y de color azul). Se marca en negro los cuatro vértices reales que se contienen.

Árbol de cajas delimitadoras

Los árboles de cajas delimitadoras son árboles binarios de cajas delimitadoras que cumplen que la caja de cada padre contiene las cajas de sus dos hijos. En el Diagrama 2.2 se muestra la estructura en forma de árbol que podría presentar una malla con 6 triángulos.

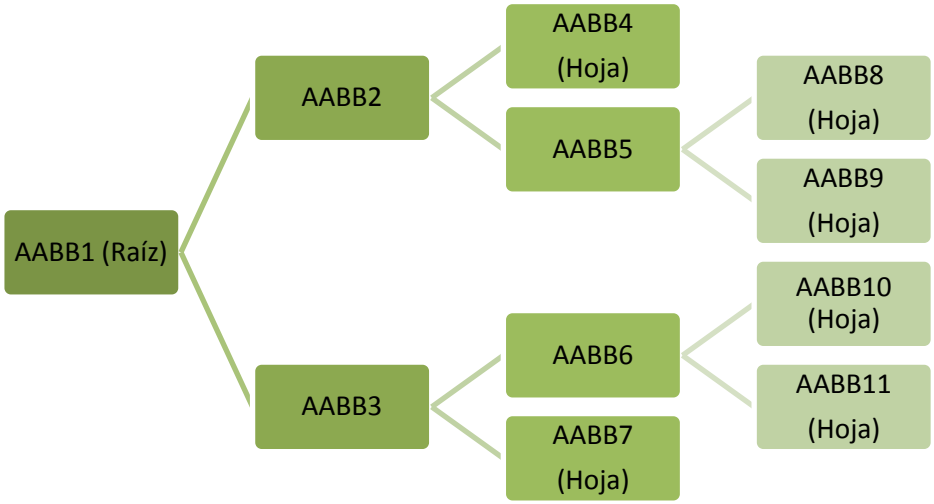


Diagrama 2.2: Árbol AABB de 4 niveles de profundidad y 6 nodos terminales

Se llama nodo a cada una de las bifurcaciones de ramas que hay en el árbol. En el Diagrama 2.2 se muestran 11 nodos. Cada nodo tiene una caja delimitadora asociada que contiene un conjunto de triángulos. Se pueden distinguir tres tipos de nodo:

a. Nodo raíz: Es aquel nodo que cuya caja delimitadora contiene toda la malla. Tiene asociada la caja delimitadora menos precisa.

b. Nodos terminales: Los nodos terminales, u *hojas*, son aquellos que únicamente contienen un elemento. En el simulador, este elemento mínimo es un triángulo.

c. Nodos intermedios: Los nodos intermedios son aquellos que ni son raíz ni terminales. Tendrán dos hijos, un nodo *izquierdo* y otro *derecho*.

Cabe destacar que a medida que se sube por el árbol, desde la raíz hasta los nodos terminales, va aumentando la precisión de la caja delimitadora. Esta característica será clave en la detección aproximada de contactos y de colisiones.

Construcción de un árbol de cajas delimitadoras

En el simulador, se implementa la construcción de un árbol de cajas delimitadoras alineadas con los ejes a partir de la malla que debe contener. El Diagrama 2.3 representa el algoritmo implementado. Es un algoritmo recursivo que se llama para construir cada uno de los nodos.

Si el número de triángulos no es superior a uno el nodo es terminal, en caso contrario se escoge el mejor eje para "partir" la caja delimitadora. Los triángulos contenidos en la caja delimitadora del nodo se reparten entre el nodo *Izquierdo* y el nodo *Derecho*, es decir, los hijos. El mejor eje es aquel que consigue un mayor equilibrio entre el número de triángulos en los dos hijos. Es decir, el mejor eje para partir la caja delimitadora es aquel que minimiza la diferencia absoluta entre el número de triángulos contenidos en el nodo hijo izquierdo y el derecho. Esto resulta en un árbol más equilibrado, por lo que la detección de contactos y colisiones se ve acelerada. El algoritmo se llama a sí mismo, de forma recursiva a todos los hijos que no sean nodos terminales. El algoritmo

está pensado para ser aplicado a toda una malla, que será contenida por un nodo raíz, y que se vaya llamando recursivamente para crear todos los nodos hasta tener todo el árbol.

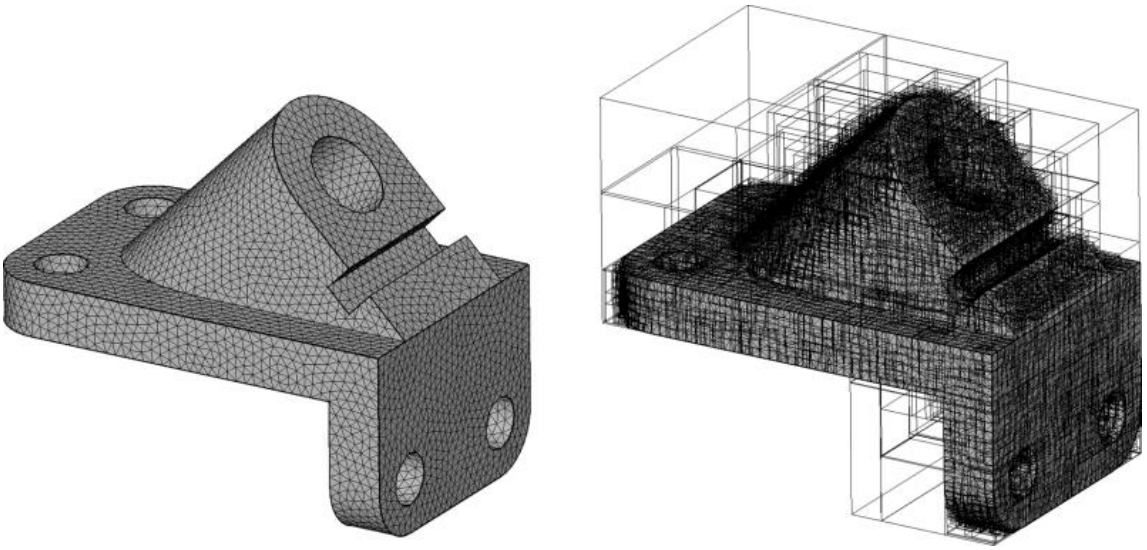


Figura 2.8: Izquierda: Anclaje mallado. Derecha: Representación de algunas de las cajas correspondientes a los nodos del árbol de AABBs

En la Figura 2.8 se observa cómo las cajas delimitadoras del árbol de una malla pueden tener medidas muy diferentes. Las cajas más grandes contienen triángulos que están muy apartados entre sí, mientras que las más pequeñas contienen un solo triángulo.

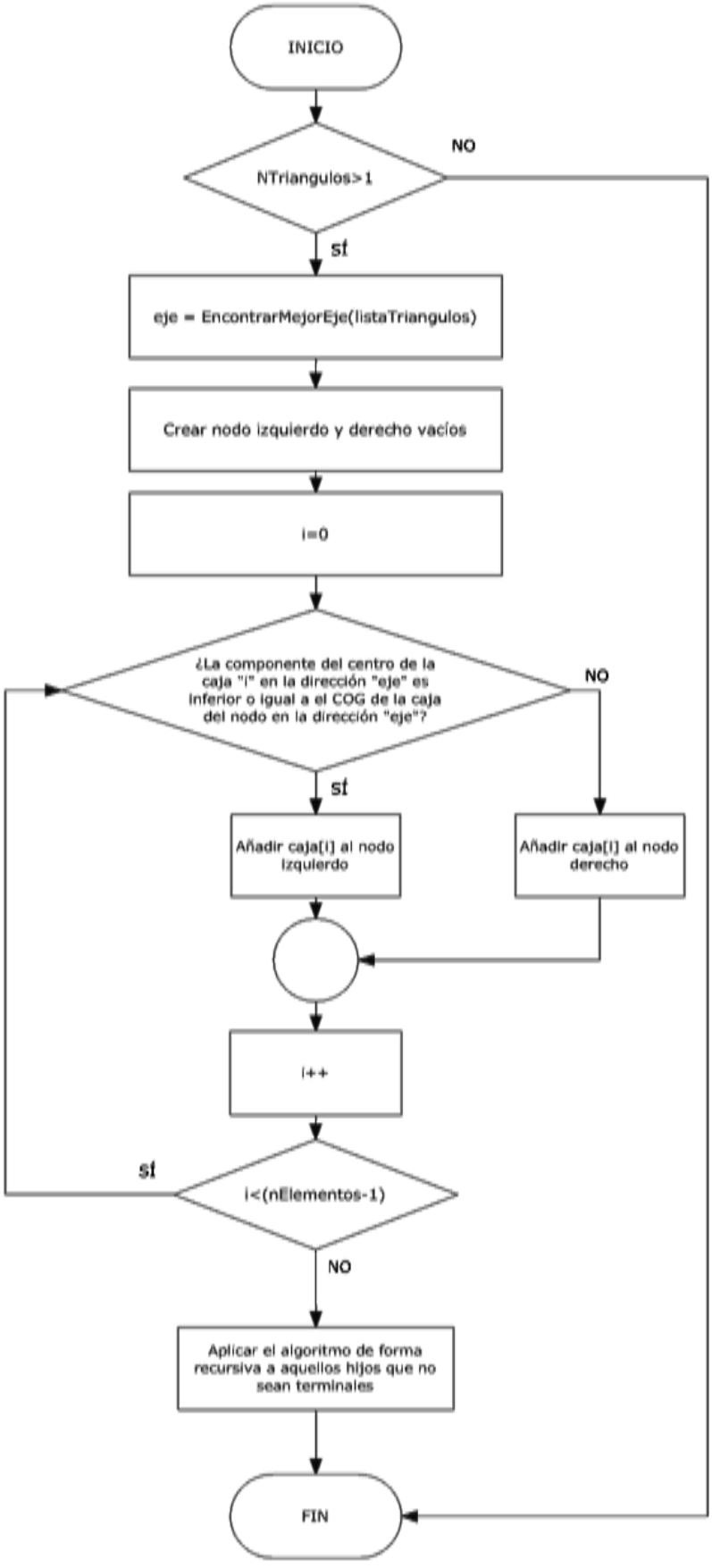


Diagrama 2.3: Algoritmo de la construcción recursiva de un árbol de AABBs a partir de una malla

Detección aproximada de contactos mediante AABBs

La detección aproximada de contactos y colisiones mediante árboles de AABB se basa en que si dos cajas delimitadoras no se solapan seguro que la geometría que contienen tampoco se intersectan.

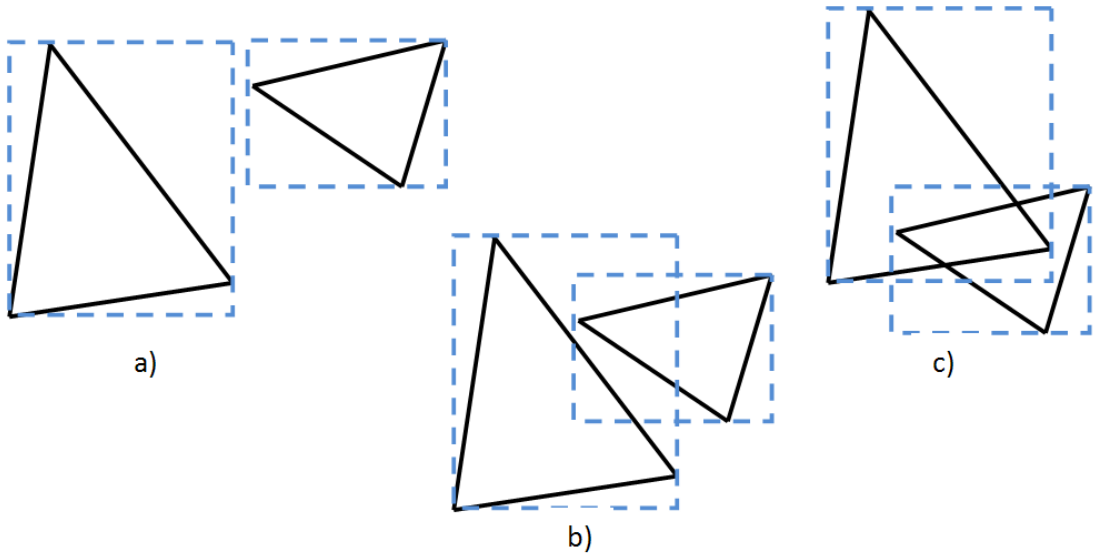


Figura 2.9: Tres posibles casos al considerar el solapamiento de cajas delimitadoras, extrapolable a tres dimensiones.

En la Figura 2.9 se muestran los tres casos posibles que pueden ocurrir al detectar contacto mediante AABBs:

a) El no solapamiento de dos cajas delimitadoras implica que no hay contacto entre las mallas que contienen.

b) Hay un solapamiento de cajas delimitadoras, pero no de la geometría que contienen.

c) Tanto las cajas delimitadoras como la geometría no se solapan.

El objetivo de la detección mediante AABB consiste en hacer un filtrado eliminando todos los casos claros de no contacto, como en el caso *a* de la Figura 2.9. Se obtendrá una *preselección* de parejas de triángulos que es *posible* que estén en contacto. Esta preselección contendrá parejas del tipo *b* y *c* de la Figura 2.9. La detección precisa posterior se encargará de filtrar los casos *b* y quedarse con los triángulos que realmente están en contacto, los del caso *c*.

El trabajar con un árbol de cajas delimitadoras permite acelerar el cálculo incluso más, pues permite comparar las *raíces* de dos árboles distintos, correspondientes a dos geometrías distintas, e ir bajando por sus ramas hasta encontrar cuáles son los nodos terminales, que contienen triángulo, que están en contacto. De esta forma no se tienen que comparar las cajas delimitadoras de todos los triángulos presentes en una escena, sino que se pueden agrupar en cajas delimitadoras más grandes y realizar una búsqueda de solapamientos más inteligente.

El Diagrama 2.4 muestra el algoritmo implementado para detectar contactos, de forma aproximada, de dos nodos *diferentes*. Es un algoritmo recursivo que crea una lista de nodos terminales que presentan solapamiento. La nomenclatura que se sigue es la siguiente:

- Caja1: Caja delimitadora correspondiente al Nodo1.
    - Izquierda1: Nodo izquierdo hijo del Nodo1.
    - Derecha1: Nodo derecho hijo del Nodo1.

- Caja2: Caja delimitadora correspondiente al Nodo2.
    - Izquierda2: Nodo izquierdo hijo del Nodo2.
    - Derecha2: Nodo derecho hijo del Nodo2.

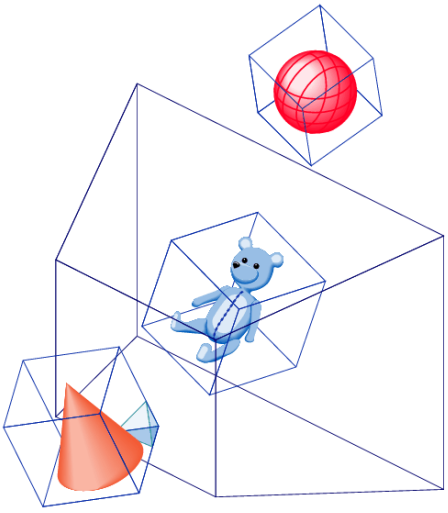


Figura 2.10: Una rápida comprobación de verificación entre la caja delimitadora del cono y de la esfera los descarta como superpuestos

A continuación se describe el proceso de detección aproximada de contactos entre dos nodos diferentes:

1. Si la caja del nodo 1 y la del nodo 2 se no se superponen el algoritmo finaliza, pues seguro que sus hijos tampoco se superpondrán.

2. En función de si los nodos son terminales o no se distinguen los casos siguientes:

    a. Los dos nodos son terminales:

        - Los nodos contienen a la pareja de triángulos que pueden estar en contacto, por lo que se añaden a la lista.

    b. Uno de los nodos es terminal:

        - Se llama recursivamente al algoritmo para buscar superposiciones entre el nodo terminal y cada uno de los hijos que no son terminales.

    c. Ningún nodo es terminal:

        - En este caso se llama recursivamente al algoritmo para buscar superposiciones entre todos los hijos. Se realizan cuatro búsquedas diferentes.

Desde un punto de vista de implementación, cabe destacar que a la función de búsqueda recursiva de superposiciones entre dos nodos tiene como parámetro una lista pasada por referencia. Esta lista se irá llenando con las parejas de triángulos que posiblemente estén en contacto.

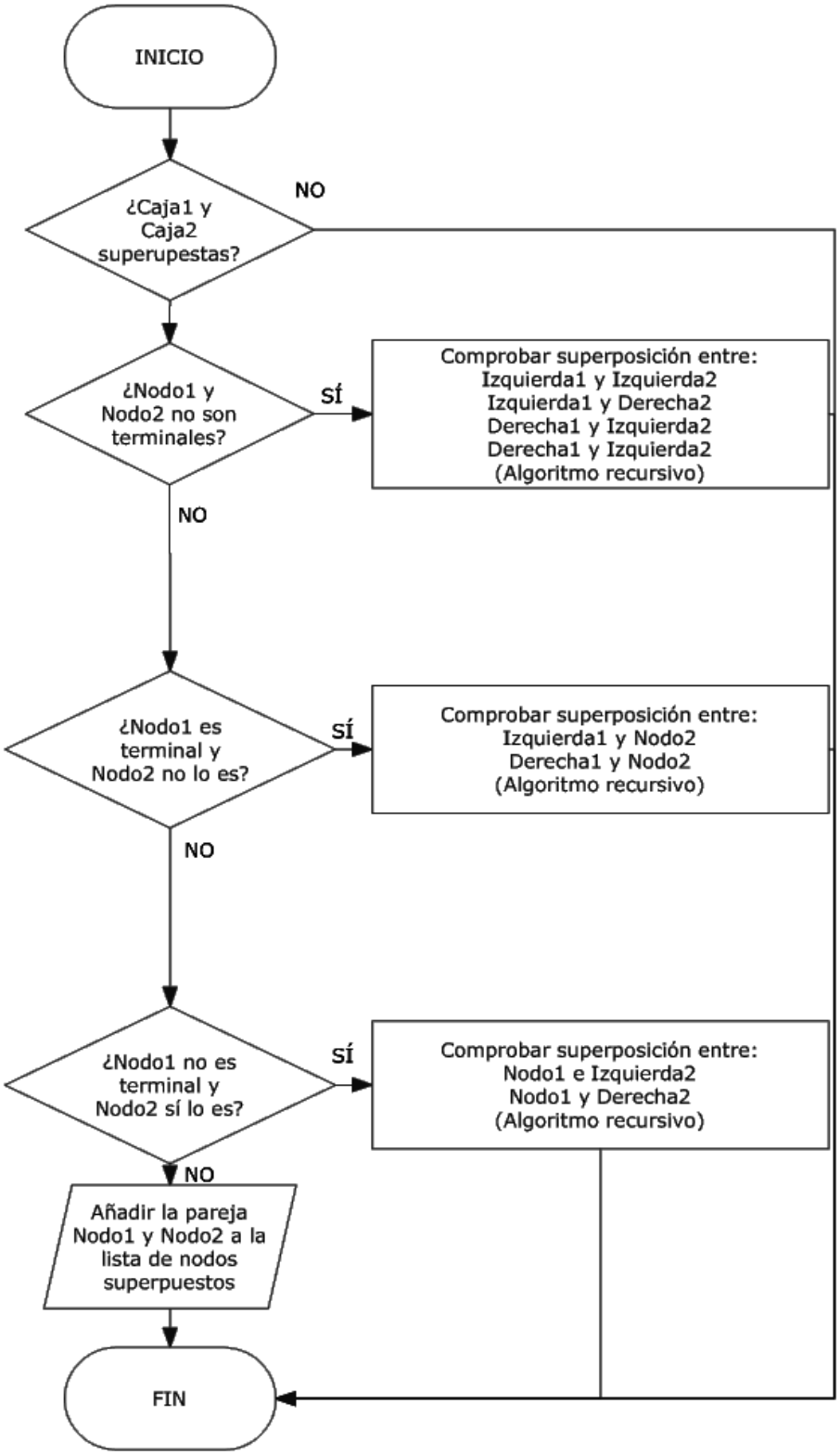


Diagrama 2.4: Algoritmo recursivo de búsqueda de superposiciones de cajas delimitadoras entre dos nodos

Diagrama 2.5: Algoritmo recursivo de detección aproximada de contactos entre los nodos de un mismo árbol.

La detección aproximada contactos de una malla consigo misma se puede realizar de forma parecida a la del Diagrama 2.4. La única diferencia consiste en que, para cada nodo, se deben buscar los contactos de sus hijos consigo mismos y los contactos de un hijo con otro. En el Diagrama 2.5 se muestra el algoritmo de detección de contactos dentro de un mismo nodo. Cabe destacar que dicho algoritmo es recursivo y hace llamadas al algoritmo del Diagrama 2.4.

Los algoritmos de creación y construcción de árboles de AABB se implementan en las clases **AABBTree** y **AABBNode**¸ para más información consulte los apartados 5.1.2 y 5.1.1 respectivamente .

Detección precisa de contactos

La detección precisa de contactos se realiza sobre un grupo de parejas de triángulos preseleccionados, aquellos que es posible que estén en contacto, por el método descrito en la sección anterior. Se realizan dos tipos de detecciones:

- Detección de contactos entre el nodo de uno de los triángulos y la superficie del triángulo opuesto. De este tipo de detecciones se realizarán seis por pareja de triángulos, una por cada nodo.
- Detección de contactos entre las aristas de un triángulo con las aristas del otro. De este tipo de detecciones se realizan nueve por pareja de triángulos, pues se debe comprobar si

hay contacto entre las aristas de un triángulo con cada arista del otro triángulo. Más adelante, se describe el algoritmo implementado para realizar este tipo de detecciones.

Este tipo de detección, cómo se explica en las secciones anteriores, es de una complejidad de cálculo significativamente superior a la detección aproximada. Por ello, la detección aproximada es necesaria para una velocidad de cálculo adecuada.

La detección precisa de contactos provee al simulador de una lista de parejas de *entidades*, nodos o aristas, que están en contacto. Posteriormente, el simulador aplicará impulsos sobre las entidades, ver apartado 2.2.5.

Contactos Nodo-Triángulo

Para determinar si un nodo y un triángulo están en contacto se realizan los siguientes pasos:

1. Se define como distancia normal para contacto nodo-triángulo al módulo del vector que va desde el nodo hasta la proyección del mismo sobre el triángulo. Esta distancia normal se corresponde a *d* en la Figura 2.11.

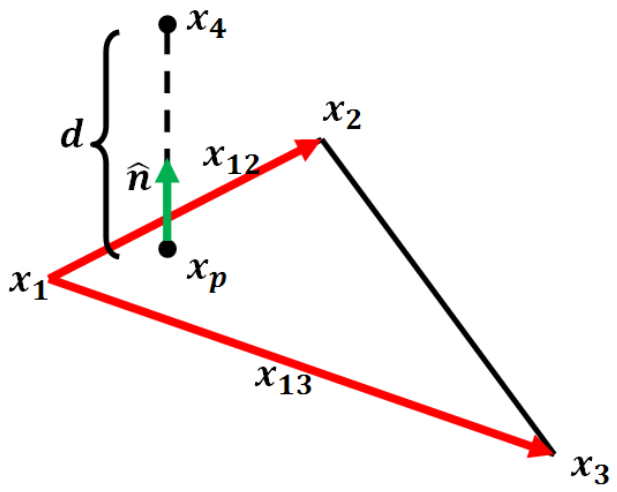


Figura 2.11: Nomenclatura utilizada para la detección de contactos entre nodos y triángulos

Primero se determina si la distancia normal es menor al grosor del triángulo, ver Figura 2.3. En caso contrario, se puede asegurar que el nodo y el triángulo no están en contacto. La distancia normal se calcula como se muestra en la ecuación (2.21).

$$d = (\mathbf{x_4} - \mathbf{x_3}) \cdot \hat{\mathbf{n}} \tag{2.21}$$

2. Si la pareja ha pasado la comprobación de proximidad anterior, se procede a calcular las coordenadas baricéntricas del punto $\mathbf{x_p}$ para determinar si la proyección del nodo sobre el plano del triángulo cae *dentro* del triángulo. En caso de que el punto $\mathbf{x_p}$ no caiga dentro del triángulo, seguro que no hay contacto entre el nodo y el triángulo. Esta comprobación es necesaria para que casos como el de la Figura 2.12 no se registren como contactos.

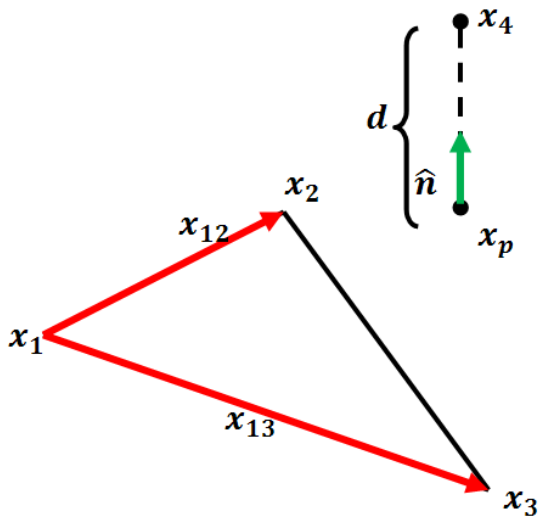


Figura 2.12: Caso en el que la segunda condición de proximidad no se cumple independientemente de que se cumpla la primera

Se denominan coordenadas baricéntricas (w_1, w_2 y w_3) de un punto, $\mathbf{x_p}$, respecto a un triángulo, con vértices $\mathbf{x_1}$, $\mathbf{x_2}$ y $\mathbf{x_3}$, a aquellas para las que se cumple la ecuación (2.22). Es equivalente a buscar las masas que deberían tener los vértices para que su centro de masas estuviese situado en $\mathbf{x_p}$.

$$\mathbf{x_p} = w_1\mathbf{x_1} + w_2\mathbf{x_2} + w_3\mathbf{x_3} \tag{2.22}$$

Para que un punto esté dentro de un triángulo, se debe cumplir que sus coordenadas baricéntricas estén entre 0 y 1 y que su suma sea igual a la unidad. Es decir, se deben cumplir las siguientes condiciones:

$$0 \leq w_i \leq 1 \ \text{ para i} = 1,2,3 \tag{2.23}$$

$$1 = w_1 + w_2 + w_3 \tag{2.24}$$

Las ecuaciones (2.30), (2.31) y (2.32) son las utilizadas para calcular las coordenadas baricéntricas del punto $\mathbf{x_p}$. A continuación se demuestran matemáticamente dichas ecuaciones. El cálculo se basa en que las coordenadas baricéntricas de la proyección del nodo sobre el triángulo se puede calcular como una aproximación por mínimos cuadrados del cálculo de coordenadas baricéntricas del nodo, suponiendo que forma parte del triángulo. Cabe destacar que para que un punto se pueda definir con coordenadas baricéntricas debe estar en el plano del triángulo. Por ello, el nodo $\mathbf{x_4}$ no se puede definir con coordenadas baricéntricas. Sin embargo, la aproximación por mínimos cuadrados para calcular las coordenadas baricéntricas del nodo $\mathbf{x_4}$ da como resultado las coordenadas baricéntricas de la proyección de dicho nodo sobre el triángulo (Bridson, Fedkiw, & Anderson, 2002). Por ello, se puede convertir la ecuación (2.22) en la (2.25).

$$\mathbf{x_1}w_1 + \mathbf{x_2}w_2 + \mathbf{x_3}w_3 = \mathbf{x_4} \tag{2.25}$$

Poniendo w_3 en función de w_1 y w_2, de la ecuación (2.24), y reordenando términos se llega a la ecuación (2.26) y, en forma matricial, (2.27).

$$\mathbf{x_{13}}w_1 + \mathbf{x_{23}}w_2 = \mathbf{x_{43}} \tag{2.26}$$

$$(\mathbf{x_{13}} \quad \mathbf{x_{23}})\begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = \mathbf{x_{43}} \tag{2.27}$$

Para realizar la aproximación por mínimos cuadrados se multiplica por la transpuesta del vector de posiciones relativas, llegando a la ecuación (2.28).

$$\begin{pmatrix} \mathbf{x_{13}} \\ \mathbf{x_{23}} \end{pmatrix}(\mathbf{x_{13}} \quad \mathbf{x_{23}})\begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} \mathbf{x_{13}} \\ \mathbf{x_{23}} \end{pmatrix}\mathbf{x_{43}} \tag{2.28}$$

$$\begin{pmatrix} \mathbf{x_{13}} \cdot \mathbf{x_{13}} & \mathbf{x_{13}} \cdot \mathbf{x_{23}} \\ \mathbf{x_{23}} \cdot \mathbf{x_{13}} & \mathbf{x_{23}} \cdot \mathbf{x_{23}} \end{pmatrix}\begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} \mathbf{x_{13}} \cdot \mathbf{x_{43}} \\ \mathbf{x_{23}} \cdot \mathbf{x_{43}} \end{pmatrix} \tag{2.29}$$

Aunque el método de Cramer no es eficiente para matrices de grandes dimensiones, sí lo es para el sistema que se presenta. Por ello, se resuelve en las ecuaciones (2.30) y (2.31) para w_1 y w_2. En la ecuación (2.32) se resuelve para w_3.

$$w_1 = \frac{\begin{vmatrix} \mathbf{x_{13}} \cdot \mathbf{x_{43}} & \mathbf{x_{13}} \cdot \mathbf{x_{23}} \\ \mathbf{x_{23}} \cdot \mathbf{x_{43}} & \mathbf{x_{23}} \cdot \mathbf{x_{23}} \end{vmatrix}}{\begin{vmatrix} \mathbf{x_{13}} \cdot \mathbf{x_{13}} & \mathbf{x_{13}} \cdot \mathbf{x_{23}} \\ \mathbf{x_{23}} \cdot \mathbf{x_{13}} & \mathbf{x_{23}} \cdot \mathbf{x_{23}} \end{vmatrix}} \tag{2.30}$$

$$w_2 = \frac{\begin{vmatrix} \mathbf{x_{13}} \cdot \mathbf{x_{13}} & \mathbf{x_{13}} \cdot \mathbf{x_{43}} \\ \mathbf{x_{23}} \cdot \mathbf{x_{13}} & \mathbf{x_{23}} \cdot \mathbf{x_{43}} \end{vmatrix}}{\begin{vmatrix} \mathbf{x_{13}} \cdot \mathbf{x_{13}} & \mathbf{x_{13}} \cdot \mathbf{x_{23}} \\ \mathbf{x_{23}} \cdot \mathbf{x_{13}} & \mathbf{x_{23}} \cdot \mathbf{x_{23}} \end{vmatrix}} \tag{2.31}$$

$$w_3 = 1 - w_1 - w_2 \tag{2.32}$$

Una vez calculadas las coordenadas baricéntricas, se determina que el nodo está en contacto con el triángulo si todas sus coordenadas baricéntricas están entre 0 y 1.

Contactos entre aristas

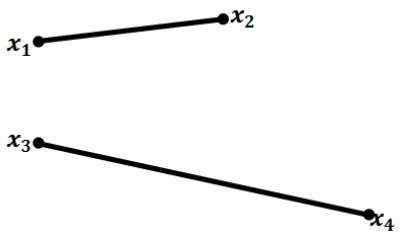


Figura 2.13: Nomenclatura de dos aristas que pueden presentar proximidad

La detección precisa de contactos arista-arista es más compleja que la de nodo-triángulo. Consta de los siguientes pasos:

1. Primero se determina cual es el punto de cada arista que está más cercano a la otra arista. Dicho de otra manera, se busca el vector con menor módulo que une a las dos aristas. Este vector tiene como origen $\mathbf{x_a}$ y como destino $\mathbf{x_b}$, como se muestra en la Figura 2.14.

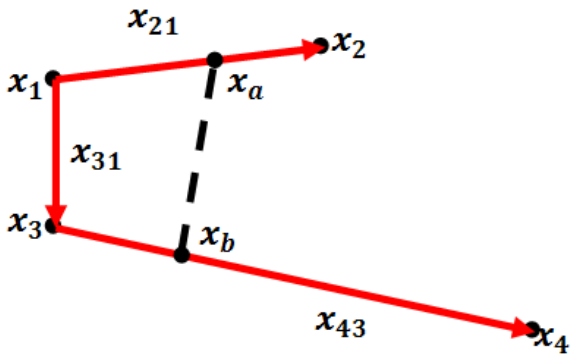


Figura 2.14: El vector xb-xa es aquel con menor módulo que une las dos aristas. Su módulo es la distancia mínima entre las dos aristas. (Se hace notar al lector que las aristas son tridimensionales)

Se definen unos parámetros *a* y *b* de tal forma que cumplan las ecuaciones (2.33) y (2.34). Cabe destacar que son como unos porcentajes que indican a qué distancia del origen del vector se encuentran los puntos más cercanos. Para que los puntos $\mathbf{x_a}$ y $\mathbf{x_b}$ se encuentren dentro de sus respectivas aristas, su valor debe estar comprendido entre 0 y 1. Sin embargo, es posible que sus valores estén fuera de este rango.

$$\mathbf{x_{21}}a + \mathbf{x_1} = \mathbf{x_a} \tag{2.33}$$

$$\mathbf{x_{43}}b + \mathbf{x_3} = \mathbf{x_b} \tag{2.34}$$

A continuación se demuestra el cálculo de *a* y *b*, necesarios para la determinación de proximidad entre aristas. De forma parecida al cálculo de detección de proximidad entre un nodo y un triángulo, se parte de una aproximación por mínimos cuadrados. Para ello, se intenta encontrar *a* y *b* suponiendo que los puntos $\mathbf{x_a}$ y $\mathbf{x_b}$ son coincidentes. Igualando las ecuaciones (2.33) y (2.34) se obtiene la ecuación (2.35). Reordenando términos y en forma matricial se obtiene la ecuación (2.37). Ésta es similar a la ecuación (2.27) y se tratará del mismo modo.

$$\mathbf{x_{21}}a + \mathbf{x_1} = \mathbf{x_{43}}b + \mathbf{x_3} \tag{2.35}$$

$$\mathbf{x_{21}}a - \mathbf{x_{43}}b = \mathbf{x_{31}} \tag{2.36}$$

$$(\mathbf{x_{21}} \quad -\mathbf{x_{43}}) \begin{pmatrix} a \\ b \end{pmatrix} = \mathbf{x_{31}} \tag{2.37}$$

A no ser que las aristas se crucen, los puntos x_a y x_b no serán coincidentes. Sin embargo, si se aplica una aproximación por mínimos cuadrados a la ecuación (2.37), se puede

encontrar aquellos puntos de la arista que están más cerca de cumplir la igualdad. Por ello, aplicando mínimos cuadrados y resolviendo para *a* y *b* se determina los puntos de las aristas, consideradas como líneas, que están más cercanos. A continuación se presenta el desarrollo. En (2.40) y (2.41) se presentan las ecuaciones finales para calcular *a* y *b*. Estas fórmulas permiten calcular *a* y *b* a partir de los datos iniciales, la posición de los cuatro vértices de las aristas.

$$\begin{pmatrix} \mathbf{x_{21}} \\ -\mathbf{x_{43}} \end{pmatrix} (\mathbf{x_{21}} \quad -\mathbf{x_{43}}) \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \mathbf{x_{21}} \\ -\mathbf{x_{43}} \end{pmatrix} \mathbf{x_{31}} \tag{2.38}$$

$$\begin{pmatrix} \mathbf{x_{21}x_{21}} & -\mathbf{x_{21}x_{43}} \\ -\mathbf{x_{21}x_{43}} & \mathbf{x_{43}x_{43}} \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \mathbf{x_{21}x_{31}} \\ -\mathbf{x_{43}x_{31}} \end{pmatrix} \tag{2.39}$$

$$a = \frac{\begin{vmatrix} \mathbf{x_{21}} \cdot \mathbf{x_{31}} & -\mathbf{x_{21}} \cdot \mathbf{x_{43}} \\ -\mathbf{x_{43}} \cdot \mathbf{x_{31}} & \mathbf{x_{43}} \cdot \mathbf{x_{43}} \end{vmatrix}}{\begin{vmatrix} \mathbf{x_{21}} \cdot \mathbf{x_{21}} & -\mathbf{x_{21}} \cdot \mathbf{x_{43}} \\ -\mathbf{x_{21}} \cdot \mathbf{x_{43}} & \mathbf{x_{43}} \cdot \mathbf{x_{43}} \end{vmatrix}} \tag{2.40}$$

$$b = \frac{\begin{vmatrix} \mathbf{x_{21}} \cdot \mathbf{x_{21}} & \mathbf{x_{21}} \cdot \mathbf{x_{31}} \\ -\mathbf{x_{21}} \cdot \mathbf{x_{43}} & -\mathbf{x_{43}} \cdot \mathbf{x_{31}} \end{vmatrix}}{\begin{vmatrix} \mathbf{x_{21}} \cdot \mathbf{x_{21}} & -\mathbf{x_{21}} \cdot \mathbf{x_{43}} \\ -\mathbf{x_{21}} \cdot \mathbf{x_{43}} & \mathbf{x_{43}} \cdot \mathbf{x_{43}} \end{vmatrix}} \tag{2.41}$$

2. En el paso anterior se ha calcularon los puntos más cercanos de las líneas infinitas que contienen a las aristas. Para saber si dos aristas están lo suficientemente cercanas para ser consideradas *próximas*, se deben buscar los puntos más cercanos que están *dentro* de las aristas.

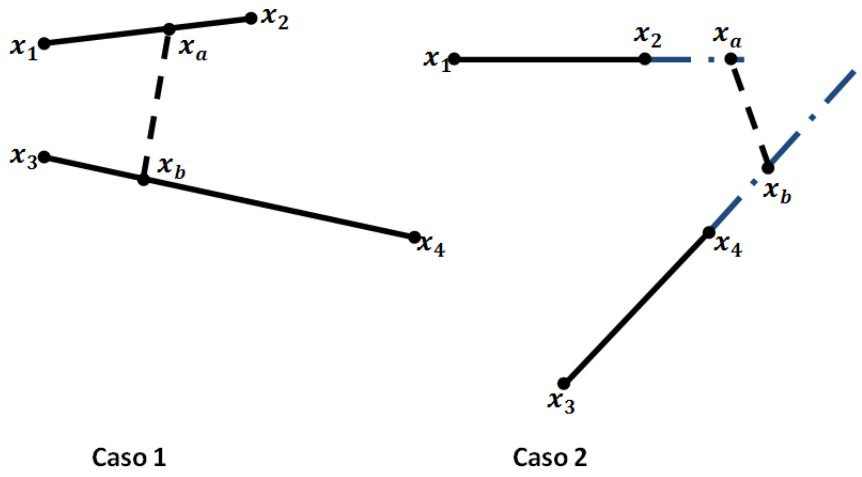


Figura 2.15: Caso 1: Los dos puntos más cercanos forman parte de las aristas. Caso 2: Alguno de los dos puntos más cercanos no forman parte de las aristas.

El Diagrama 2.6 muestra todo el algoritmo que se sigue para determinar si hay contatcto entre una pareja de aristas. El cálculo de *a* y *b* ya se ha presentado en las ecuaciones (2.40) y (2.41). Si *a* y *b* están entre 0 y 1, los puntos más cercanos se encuentran dentro de las

aristas. Por ello, se calcula el vector **d** que une los puntos más cercanos, según la ecuación (2.42). Si el módulo de **d** es inferior al grosor de la tela, se determina que las aristas están en contacto.

$$\mathbf{d} = \mathbf{x_b} - \mathbf{x_a} = (b\mathbf{x_{43}} + \mathbf{x_3}) - (a\mathbf{x_{21}} + \mathbf{x_1}) = b\mathbf{x_{43}} - a\mathbf{x_{21}} + \mathbf{x_{31}} \qquad (2.42)$$

En caso de que *a* y *b* no estén entre 0 y 1 se deben encontrar los puntos más cercanos que sí formen parte de las aristas. Para ello se definen otros dos parámetros *a2* y *b2* de la siguiente forma:

Si $a < 0$, *a2* es 0

Si $a > 1$, *a2* es 1

Si $0 \leq a \leq 1$, *a2* es *a*

Si $b < 0$, *b2* es 0

Si $b > 1$, *b2* es 1

Si $0 \leq b \leq 1$, *b2* es *b*

Si $\mathbf{x_a}$ o $\mathbf{x_b}$ caen fuera de la arista que le corresponde, se busca un punto $\mathbf{x_{a2}}$ o $\mathbf{x_{b2}}$, que es el punto más cercano a $\mathbf{x_a}$ o $\mathbf{x_b}$, según corresponda, que forme parte de la arista. Evidentemente, esto implica que $\mathbf{x_{a2}}$ o $\mathbf{x_{b2}}$ son $\mathbf{x_a}$ o $\mathbf{x_b}$ o uno de los extremos de las aristas, ver la Figura 2.16. Estos puntos no tienen porque ser los más cercanos, pero sí se sabe que el punto que más se ha desplazado seguro que es parte de la solución (Bridson, Fedkiw, & Anderson, 2002).

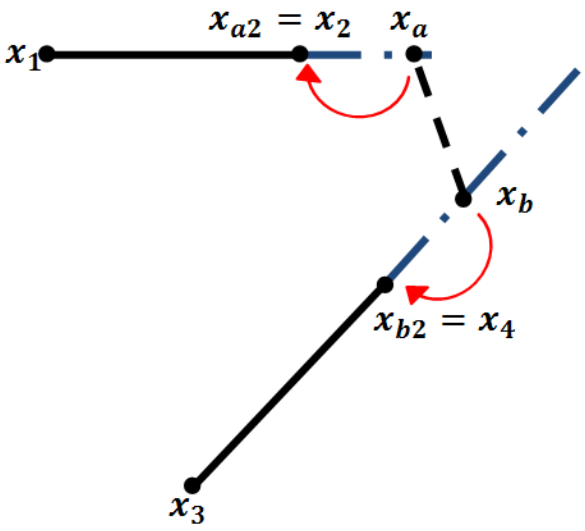


Figura 2.16: Si x_a o x_b no forman parte de sus correspondientes aristas, se busca el extremo de la arista. Estos nuevos puntos son x_{a2} y x_{b2}.

El vector $\mathbf{L_1}$ representa el desplazamiento de $\mathbf{x_{a2}}$ a $\mathbf{x_a}$ y el vector $\mathbf{L_2}$ representa el desplazamiento de $\mathbf{x_{b2}}$ a $\mathbf{x_b}$. $\mathbf{L_1}$ y $\mathbf{L_2}$ se calculan con las ecuaciones (2.43) y (2.44) respectivamente.

$$\mathbf{L_1} = \mathbf{x_a} - \mathbf{x_{a2}} = \mathbf{x_{21}} \cdot (a - a_2) \qquad (2.43)$$

$$\mathbf{L_2} = \mathbf{x_b} - \mathbf{x_{b2}} = \mathbf{x_{43}} \cdot (\mathrm{b} - \mathrm{b}_2) \tag{2.44}$$

Se compara el módulo de estos dos vectores para determinar cual es el punto, $\mathbf{x_a}$ o $\mathbf{x_b}$, que más se ha movido. Por lo tanto, hay dos posibilidades:

I. Si $\mathbf{x_a}$ es el punto que más se ha movido se fija éste, pues se sabe que es parte de la solución, y se busca que punto de la otra arista es el más cercano a $\boldsymbol{x_{a2}}$. Esto se puede hacer fácilmente partiendo de la ecuación (2.35). Se pueden reordenar los términos para llegar a la ecuación (2.45). Cabe destacar que $\mathbf{x_{43}}$ es un vector de 3 componentes, por lo que se trata de un sistema de ecuaciones con tres ecuaciones y una incógnita. Por ello, se debe multiplicar por la transpuesta de $\mathbf{x_{43}}$, por la izquierda, a cada lado de la igualdad. De esta forma, se aplica el método de mínimos cuadrados y se obtiene b_2, ecuación (2.46).

$$\mathbf{x_{43}} b_2 = \mathbf{x_{13}} + \mathbf{x_{21}} a_2 \tag{2.45}$$

$$b_2 = \frac{\mathbf{x_{43}} \cdot (\mathbf{x_{13}} + \mathbf{x_{21}} a_2)}{\mathbf{x_{43}} \cdot \mathbf{x_{43}}} \tag{2.46}$$

Es posible que b_2 se encuentre fuera de la arista 2. En tal caso, se mueve al extremo de la arista más cercano. De forma análoga a la anterior:

Si $b2 < 0$, $b2$ es 0

Si $b2 > 1$, $b2$ es 1

II. En caso de que $\mathbf{x_b}$ sea el punto que más se ha movido, se realiza un proceso similar al anterior. Se aplica la ecuación (2.47) para calcular a_2 y se ajusta a uno de los extremos de las aristas en caso de que sea necesario.

$$\mathbf{a_2} = \frac{\mathbf{x_{21}} \cdot (\mathbf{x_{31}} + \mathbf{x_{43}} \mathbf{b_2})}{\mathbf{x_{21}} \cdot \mathbf{x_{21}}} \tag{2.47}$$

Ahora que ya se tiene seguridad sobre que puntos de las aristas son los más cercanos, se calcula la distancia entre los mismos y se compara con el grosor de la tela. Si están a una distancia menor a dicho grosor, se sabe que las aristas están en contacto.

Este algoritmo se aplica a todas las aristas que han sido preseleccionadas por el detector basado en *AABB* y genera una lista de aristas que *seguro* que están en contacto. Sobre éstas, posteriormente, se aplicarán impulsos de repulsión, ver el apartado 2.2.5.

El algoritmo de detección precisa de contactos se implementa en la clase **WorldProximityDetector,** documentada en el apartado 5.1.17. Ésta llama a la clase **AABBTree** para detectar contactos de forma aproximada, con cajas delimitadoras.

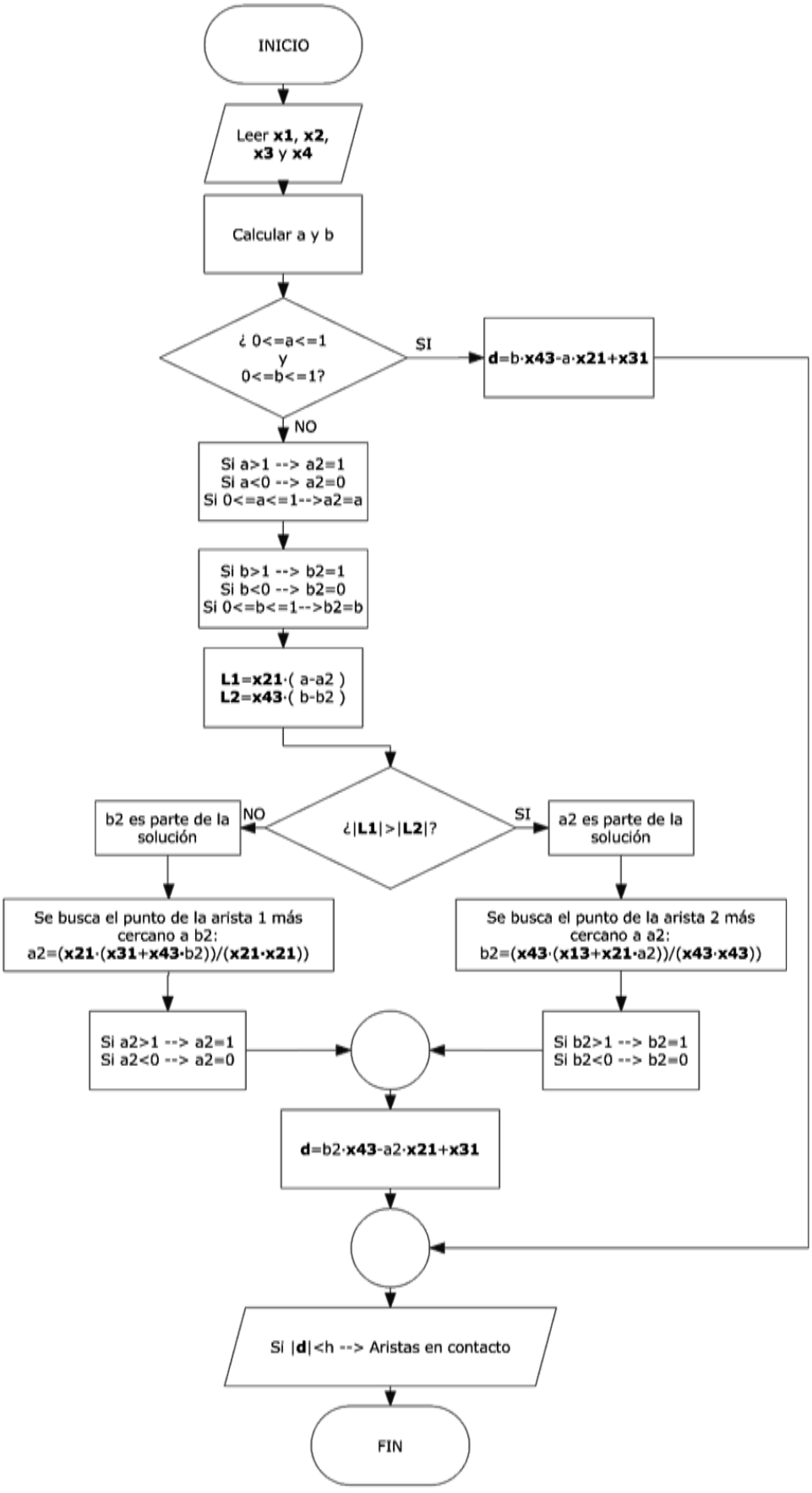


Diagrama 2.6: Diagrama de flujo de determinación de contacto entre una pareja de aristas

2.2.4 Detección de colisiones

La detección de colisiones consiste en los mismos pasos que la detección de contactos, sólo que con algunas diferencias:

I. Primero hay una detección aproximada de colisiones que permite descartar aquellos elementos que seguro que no colisionan. Se genera una lista de candidatos preseleccionados.

II. Se desglosa cada candidato preseleccionado en un conjunto de comprobaciones de colisión de tipo nodo-triángulo y arista-arista. Se genera una lista final con parejas de aristas, nodos y triángulos que colisionan durante el paso de simulación que se está llevando a cabo. A las entidades que colisionen se les aplicará posteriormente unos impulsos que evitarán la colisión, ver apartado 2.2.5.

Cabe destacar que para realizar la detección de colisiones se parte de una *estimación* del estado de las mallas al final del paso de simulación. Puesto que la resolución de colisiones es un proceso iterativo, esta estimación puede haberse calculado únicamente a partir de la dinámica interna de la ropa o a partir de la dinámica interna modificada con impulsos debido a colisiones detectadas en iteraciones previas.

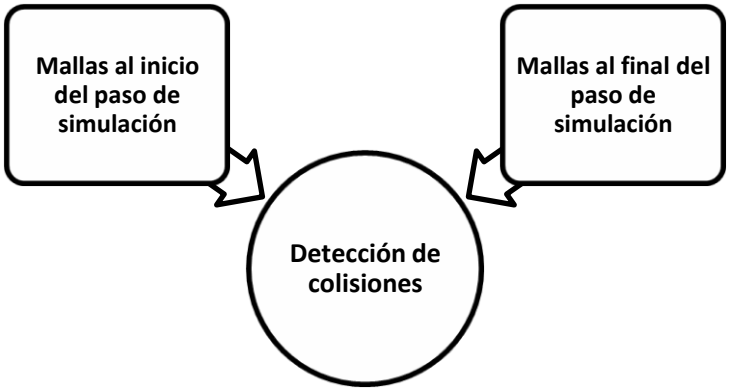


Diagrama 2.7: Se requiere una estimación de las mallas al final del paso de simulación para detectar colisiones

Detección aproximada de colisiones

A diferencia de los contactos, que se producen *al principio* del paso de simulación, las colisiones se producen *durante* el paso de simulación. Por ello, las cajas delimitadoras que se aplican en la detección aproximada de colisiones, engloban las mallas al inicio y al final del paso de simulación. Las conclusiones que se pueden extraer de la intersección de cajas delimitadoras son las mismas que en el caso de contactos:

- Si dos cajas delimitadoras no se solapan *seguro* que los elementos que contienen no colisionan durante el paso de simulación. Por ello no es necesario hacer una comprobación de colisión precisa con un coste de cálculo muy superior al de la detección aproximada.
- Si dos cajas delimitadoras se solapan es *posible* que los elementos que contienen colisionen. Posteriormente, se realiza una detección precisa.

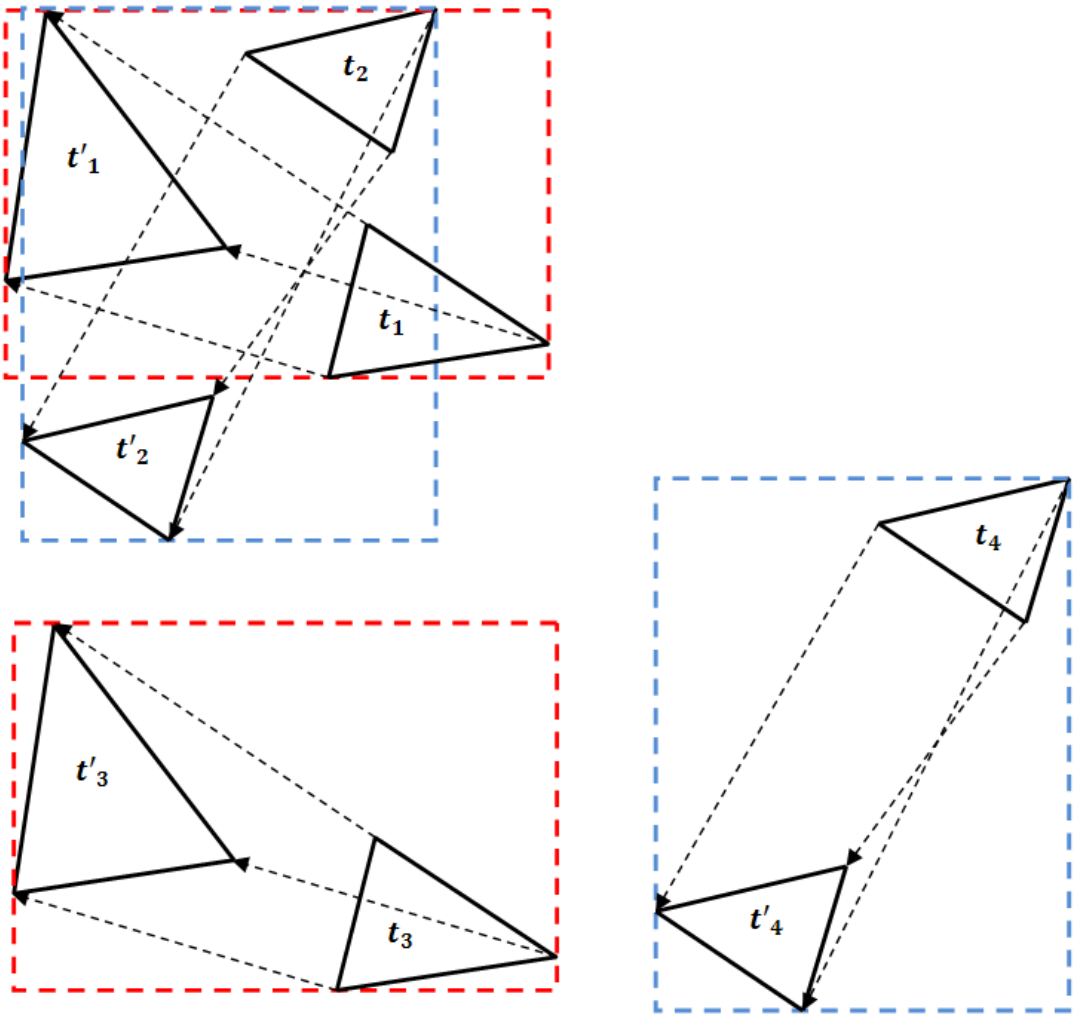


Figura 2.17: Las cajas delimitadoras engloban tanto a los triángulos al principio como al final del paso de simulación. En esta figura es imposible que los triángulo 3 y 4 colisionen, pues sus cajas delimitadoras no se solapan. Sin embargo, los triángulos 1 y 2 sí es posible que colisionen.

Los árboles de cajas delimitadoras utilizados en detección de colisiones se generan con un algoritmo similar al descrito en el Diagrama 2.3. La diferencia es que los nodos terminales deben contener el triángulo en su posición al principio y al final del paso de simulación. Para ello, no se consideran los triángulos como unidades mínimas indivisibles, sino las parejas de triángulos. Para ello, se denomina *elemento* a la pareja de triángulos dada por la posición inicial y la final de un mismo triángulo durante un paso de simulación. Antes de construir el árbol de cajas delimitadoras, se debe generar la lista de elementos que debe contener.

La búsqueda de parejas de nodos terminales que se solapan se realizará con los mismos algoritmos que en el caso de detección de contactos, ver Diagrama 2.4 para nodos terminales de mallas diferentes y Diagrama 2.5 para nodos terminales de una misma malla.

El algoritmo de construcción y detección de colisiones con árboles de AABB también se implementa en la clase **AABBTree**, documentada en el apartado 5.1.2.

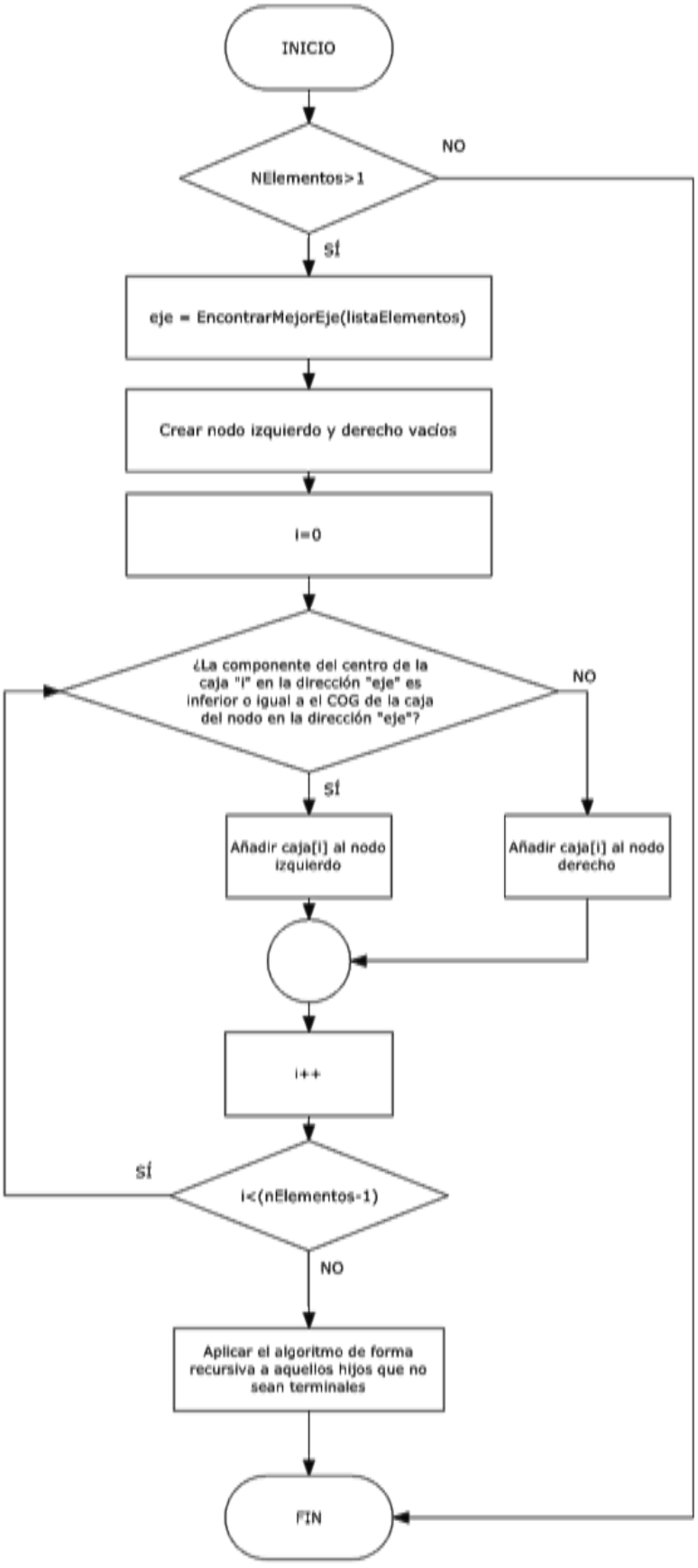


Diagrama 2.8: Diagrama de flujo de la construcción del nodo de un árbol de cajas delimitadoras para la detección aproximada de colisiones. Se trabaja con elementos, parejas de triángulos.

Detección precisa de colisiones

De forma similar a la detección precisa de contactos, se realizan detecciones precisas de colisiones de tipo nodo-triángulo y arista-arista. Se parte de 4 triángulos, que corresponden al estado de 2 triángulos al inicio y al final del paso que se está simulando. Por ello se realizan las siguientes comprobaciones:

- 6 comprobaciones de colisión nodo-triángulo. Una por cada nodo, o vértice, de los dos triángulos que se desplazan durante el lapso de tiempo simulado.
- 9 comprobaciones de colisión arista-arista. Una por cada pareja de aristas de los dos triángulos diferentes.

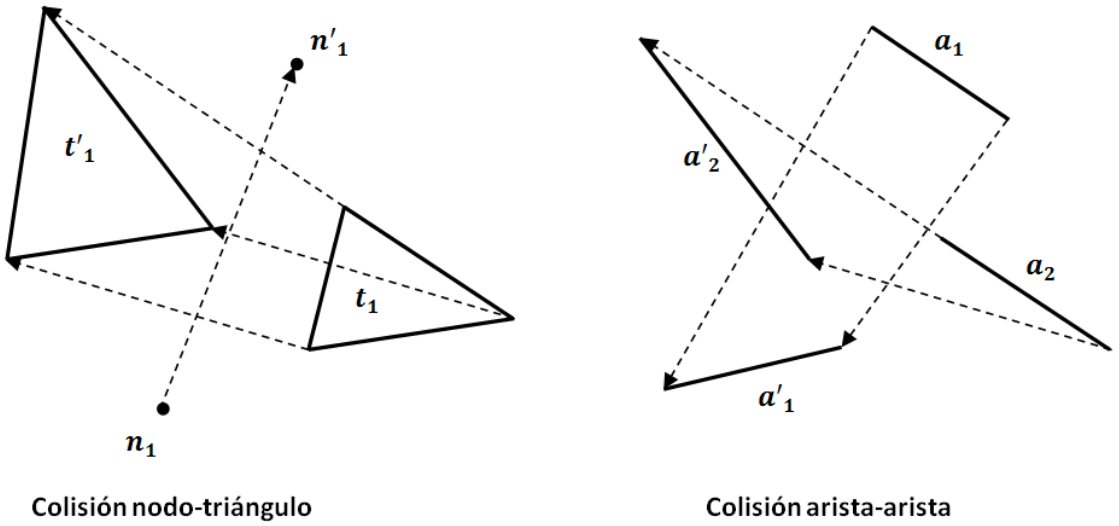


Figura 2.18: Se diferencian dos tipos de colisión: colisiones nodo-triángulo y colisiones arista-arista

Detección de colisiones Nodo-Triángulo

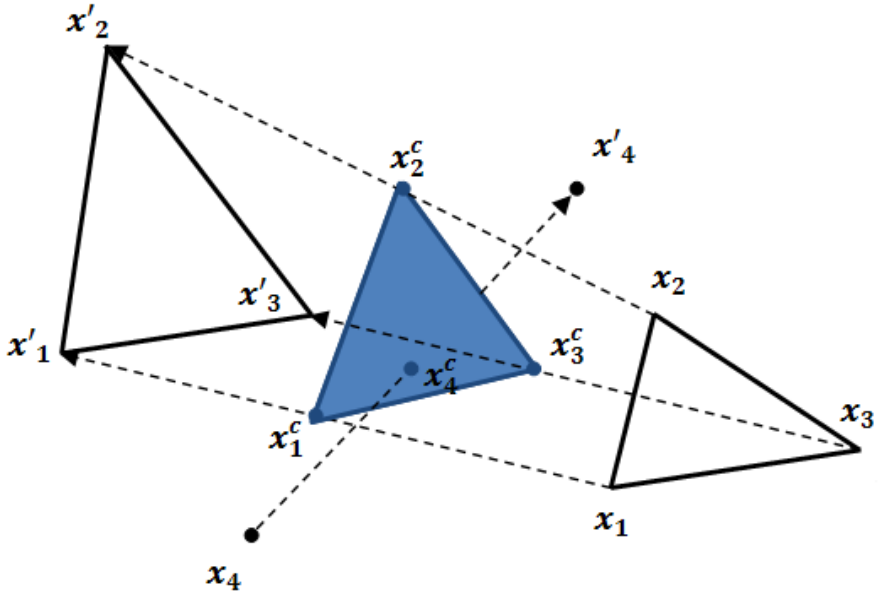


Figura 2.19: Nomenclatura utilizada para determinación de colisión nodo-triángulo

En la Figura 2.19 se muestra parte de la nomenclatura que se sigue en la determinación de colisiones nodo-triángulo. Se parte de los siguientes datos:

- $\mathbf{x_1}$, $\mathbf{x_2}$ y $\mathbf{x_3}$: Vectores de posición de los tres vértices del triángulo al inicio del paso que se está simulando.
- $\mathbf{x_4}$: Vector de posición del nodo al inicio del paso que se está simulando.
- $\mathbf{x_1'}$, $\mathbf{x_2'}$ y $\mathbf{x_3'}$: Vectores de posición de los tres vértices del triángulo al final del paso que se está simulando. Estas posiciones son estimadas, pudiendo dar lugar a colisiones.
- $\mathbf{x_4'}$: Vector de posición del nodo al final del paso que se está simulando. Esta posición es estimada, pudiendo dar lugar a colisiones.

Se dice que el nodo y el triángulo colisionan en el paso simulado si cumplen las siguientes condiciones:

I. En el lapso de tiempo simulado, y suponiendo que los nodos siguen un movimiento rectilíneo, hay un momento en el que los cuatro nodos, $\mathbf{x^c_1}$ a $\mathbf{x^c_4}$, son *coplanarios*.

II. En el momento en el que los cuatro nodos son coplanarios el nodo $\mathbf{x^c_4}$ se encuentra dentro del triángulo $\mathbf{x^c_1}$ a $\mathbf{x^c_3}$.

A continuación se detalla el algoritmo utilizado para determinar si se cumplen estas dos condiciones.

Primero, se calculan las velocidades suponiendo que los nodos se mueven rectilíneamente a partir de la ecuación (2.48), en la que t_s es el lapso de tiempo simulado en el paso calculado.

$$\mathbf{v_i} = \frac{\mathbf{x_i'} - \mathbf{x_i}}{t_s} \quad i = 1,2,3,4 \tag{2.48}$$

Si los nodos llegan a ser coplanarios en algún momento, se cumple que en ese momento los 3 vectores que van desde el nodo 1 al resto de los nodos son coplanarios. Por ello se debe cumplir la ecuación (2.49). Esta ecuación se puede poner en función de las posiciones iniciales, las velocidades calculadas y el tiempo en el que se produce la colisión, en caso de que lo haga, dando lugar a la ecuación (2.50) (Bridson, Fedkiw, & Anderson, 2002).

$$\mathbf{x^c_{21}} \times \mathbf{x^c_{31}} \cdot \mathbf{x^c_{41}} = 0 \tag{2.49}$$

$$\begin{gathered} p_3(t) = (\mathbf{x_{21}} + t\mathbf{v_{21}}) \times (\mathbf{x_{31}} + t\mathbf{v_{31}}) \cdot (\mathbf{x_{41}} + t\mathbf{v_{41}}) \\ p_3(t_c) = 0 \end{gathered} \tag{2.50}$$

La ecuación (2.50) tiene como incógnita el tiempo t_c en el que se produce la colisión. Para resolver la ecuación se debe desarrollar la ecuación. Por brevedad, no se presenta la ecuación completamente desarrollada. Sin embargo, se indica que es un polinomio de tercer grado con la forma mostrada en la ecuación (2.51).

$$At_c^3 + Bt_c^2 + Ct_c + D = 0 \tag{2.51}$$

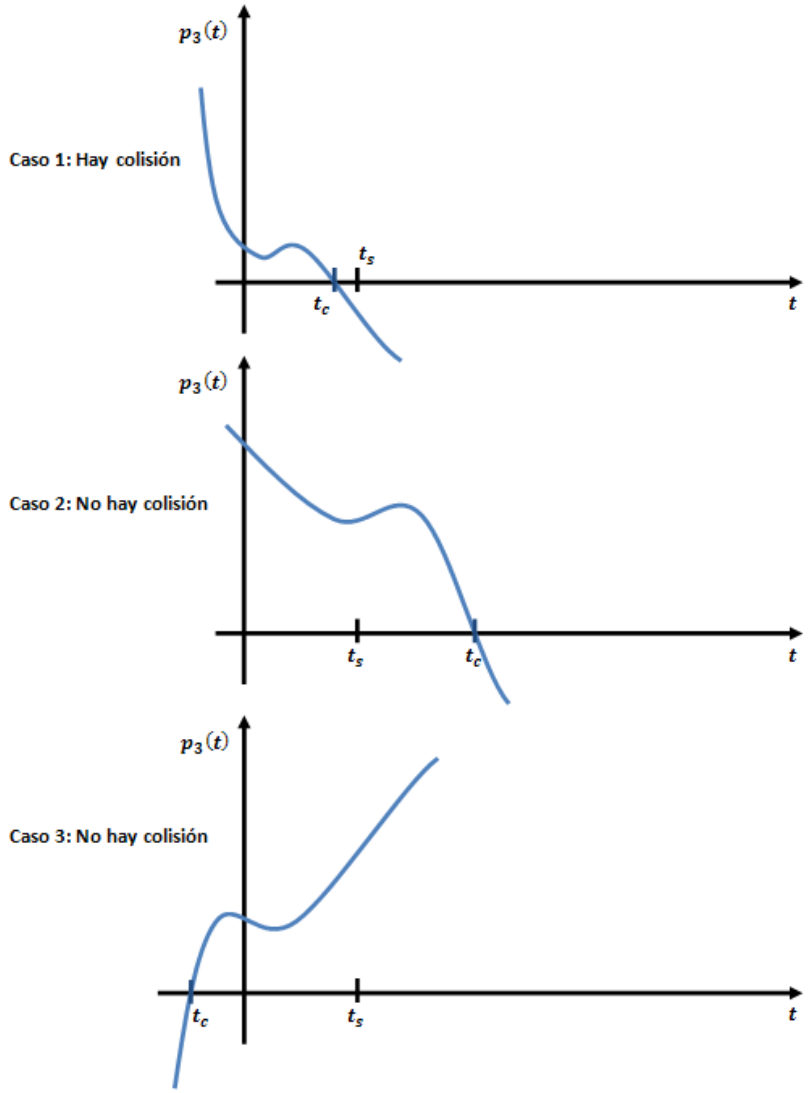


Figura 2.20: La raíz se puede encontrar en tres zonas. Únicamente en el caso 1 puede haber colisión, pues la raíz se encuentra dentro del intervalo simulado.

Para facilitar la búsqueda de la raíz más pequeña que queda dentro del intervalo de tiempo simulado, se calcula primero el máximo y el mínimo del polinomio. Esto se puede hacer fácilmente encontrando las raíces de la derivada, ecuación (2.52). Se utilizan los mismos parámetros que en la ecuación (2.51). Los tiempos en los que ocurre el mínimo y el máximo se pueden calcular a partir de la ecuación (2.53), que es la ecuación (2.52) resuelta. Se denomina t_{e1} al tiempo mínimo y t_{e2} al máximo.

$$3At_e^2 + 2Bt_e + C = 0 \tag{2.52}$$

$$t_e = \frac{-2B \pm \sqrt{4B^2 - 12AC}}{6A} \tag{2.53}$$

Es útil conocer el máximo y el mínimo de la función para saber dónde buscar las raíces. Se distinguen cuatro casos:

i. $t_{e1} < 0 < t_s < t_{e2}$:

En este primer caso, tanto el máximo como el mínimo del polinomio se encuentran fuera del intervalo de tiempo simulado. Por ello, la raíz se buscará en todo el intervalo de tiempo simulado en el paso actual. 0 y t_s serán los parámetros pasados a la función de búsqueda de raíces que delimitarán el rango de búsqueda.

Cabe destacar que en la Figura 2.21 la función no tiene raíces dentro del intervalo. Esto se detectará posteriormente, y fácilmente, al ver que el producto de las imágenes en 0 y t_s es positiva. En este caso concreto, se aseguraría que los cuatro nodos no llegan a ser coplanarios en el intervalo de tiempo simulado en este paso.

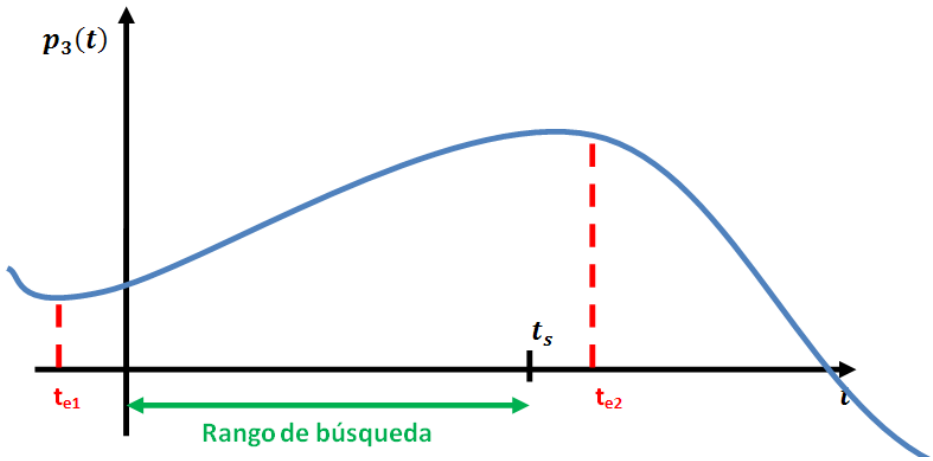


Figura 2.21: Mínimo y máximo del polinomio está fuera del intervalo de tiempo.

ii. $t_{e1} < 0 < t_{e2} < t_s$:

En este segundo caso, únicamente el máximo o mínimo en t_{e2} se encuentra dentro del intervalo de tiempo simulado. Por ello, se empezará a buscar una raíz entre 0 y t_{e2} y, en caso de que no se encuentre, se buscará otra entre t_{e2} y t_s.

Estos dos rangos de búsqueda se muestran con color verde en la Figura 2.22. En este caso en particular, la búsqueda en el primer rango ya daría una solución, un tiempo en el que los cuatro nodos han estado en el mismo plano, por lo que no se necesitaría buscar en el segundo rango.

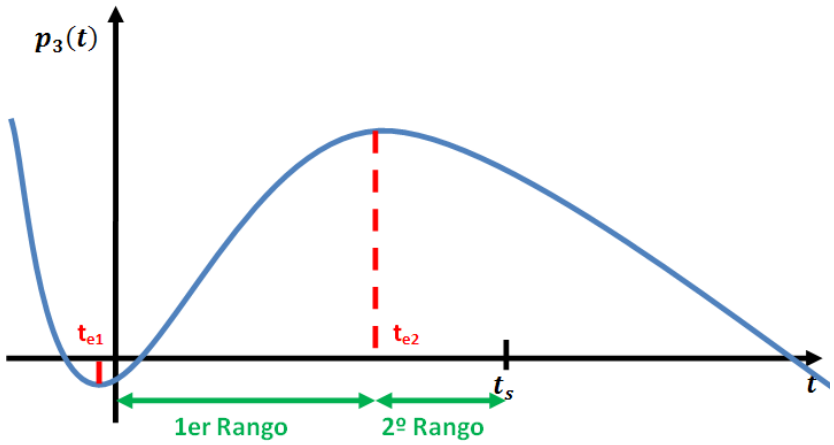


Figura 2.22: Sólo el máximo (t_{e2}) está dentro del intervalo de simulación.

iii. $0 < t_{e1} < t_s < t_{e2}$**:**

Este tercer caso es parecido al segundo, con la diferencia de que sólo el máximo o mínimo en t_{e1} se encuentra en intervalo de tiempo simulado. Por ello, se empieza una búsqueda de raíces entre 0 y t_{e1} y, en caso de que no se halle raíz alguna, se busca entre t_{e1} y t_s.

La Figura 2.23 muestra un ejemplo de este caso en el que la búsqueda en el primer rango ya daría una solución. Sin embargo, cabe destacar que un movimiento rectilíneo de los nodos nunca dará lugar a un polinomio así, pues implicaría que en más de un instante los cuatro nodos son complanarios.

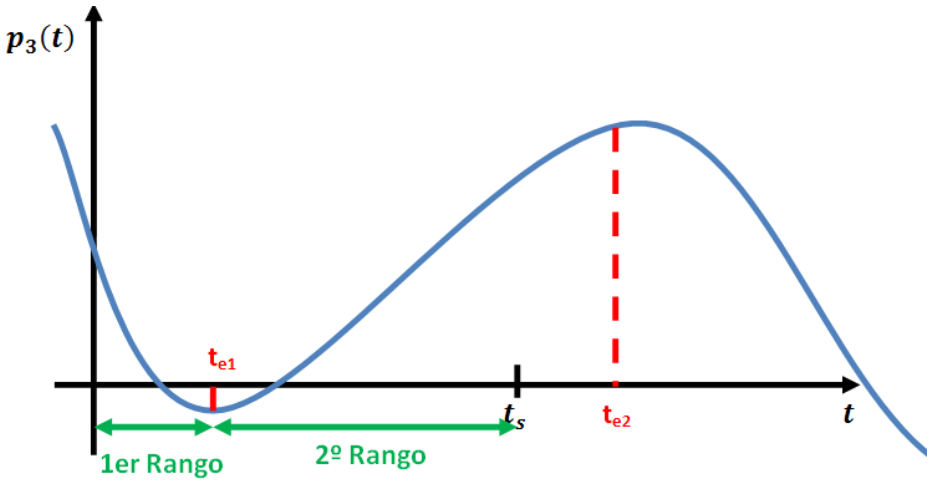


Figura 2.23: Sólo el mínimo (t_{e1}) se encuentra dentro del intervalo de simulación.

iv. $0 < t_{e1} < t_{e2} < t_s$**:**

En este último caso, tanto el máximo como el mínimo se encuentran en el intervalo de tiempo simulado. Se buscarán raíces en los intervalos 0 a t_{e1}, t_{e1} a t_{e2} y t_{e2} a t_s en este orden. Cuando se haya encontrado una raíz se dejará de buscar en los siguientes rangos.

La Figura 2.24 muestra un ejemplo de este caso en el que la búsqueda de raíz en el primer rango ya daría resultado.

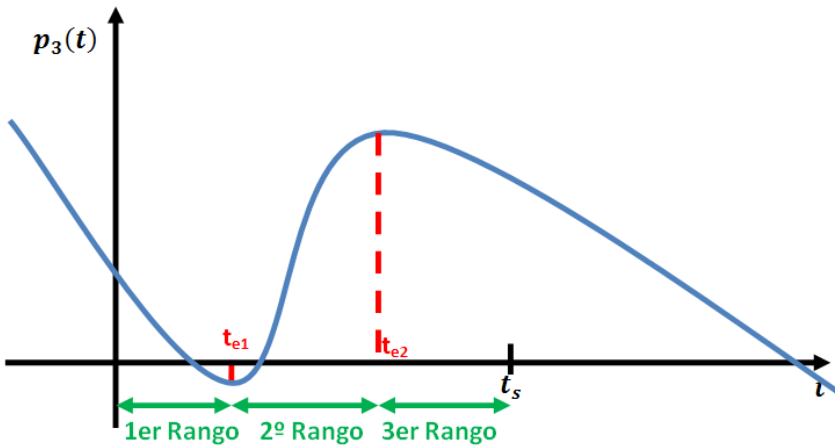


Figura 2.24: Tanto el mínimo (te1) como el máximo (te2) se encuentran dentro del intervalo de simulación.

El método seleccionado para encontrar la raíz positiva más pequeña que esté dentro del intervalo de simulación es el de la secante. Se podría haber escogido un método concreto para polinomios, como el de Bairstow. Sin embargo, el método de la secante presenta la ventaja de sólo calcular una raíz si se cree que está dentro del intervalo de simulación. Es decir, omite las raíces que seguro que no son útiles, ahorrando tiempo de cálculo. De todas formas, cabe destacar que un perfilado del programa final ha dado como resultado que el tiempo empleado en determinar si los nodos llegan a ser coplanarios es muy pequeño. Por ello, encontrar un método de búsqueda de raíces más eficiente no aceleraría significativamente el tiempo de cálculo.

Para aplicar el método de la secante, se ha adaptado la función *rtsec* del *Numerical Recipes in C* (H. Press, A. Teukolsky, T. Vetterling, & P. Flannery, 1995) para que resuelva polinomios de tercer grado. A continuación se presenta la función modificada, *g3PolRtsec*, que acepta como parámetro adicional un puntero *poly* a una lista con los 4 coeficientes del polinomio.

```
bool WorldCollisionDetector::g3PolRtsec ( const float * const
poly,float ( *func ) ( const float * const, const float ), const
float x1, const float x2, const float xacc,float &rootSolution )
{
 int j;
 float fl,f,dx,swap,xl,rts;

 fl= ( *func ) ( poly,x1 );
 f= ( *func ) ( poly,x2 );

 if ( ( fl*f ) <0 )
 {
       if ( fabs ( fl ) <fabs ( f ) )
       {
             rts=x1;
             xl=x2;
             swap=fl;
             fl=f;
             f=swap;
       }
       else
       {
             xl=x1;
             rts=x2;
       }
       for ( j=1;j<=MAXIT;j++ )
       {
             dx= ( xl-rts ) *f/ ( f-fl );
             xl=rts;
             fl=f;
             rts+=dx;
             f= ( *func ) ( poly,rts );
             if ( fabs ( dx ) <xacc ||f==0.0 )
             {
                   rootSolution=rts;

                   return true; //convergence
             }
       }
       std::cout<<"\nPolynomial coefficients passed to rtsec: "<<
       poly[3]<<" "<<poly[2] <<" "<< poly[1]<<" "<< poly[0]<<"\n";
       nrerror ( "Maximum number of iterations exceeded in
       rtsec." );
       return false;
 }
```

```
    else
    {
          return false;
    }
}
```

En el Gráfico 2.2, se presenta un caso real de simulación en el que los nodos son coplanarios dentro del intervalo de tiempo simulado. En el Gráfico 2.3 la raíz queda fuera del intervalo simulado, por lo que los nodos no llegan a ser coplanarios.

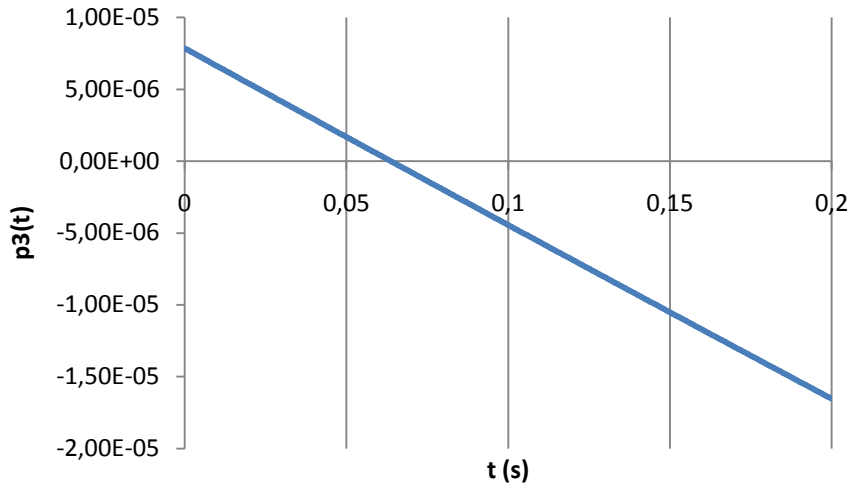


Gráfico 2.2: Caso real en el que los nodos llegan a ser coplanarios durante el intervalo de simulación, que es de 0,1s. p3(t) = 1,034E-5t^2- 1,24E-4t + 7,84E-6.

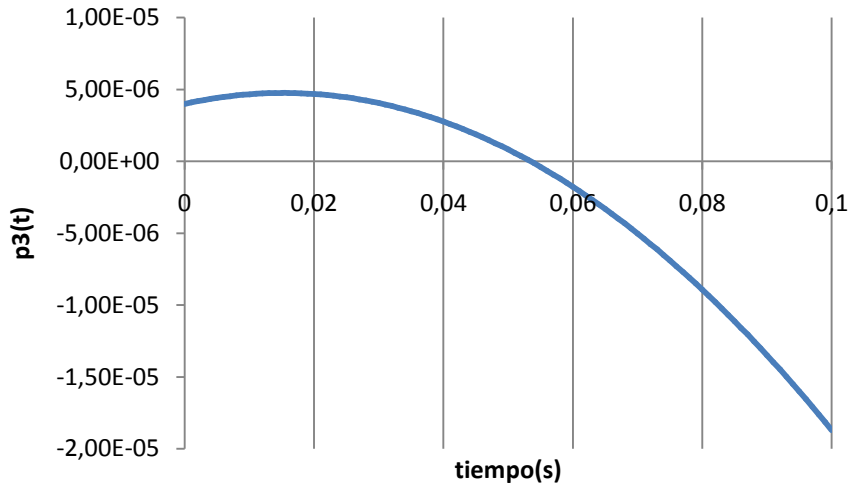


Gráfico 2.3: Caso real en el que los nodos nollegan a ser coplanarios durante el intervalo de simulación, que es de 0,05s. p3(t) = -3,27E-3t^2 + 1E-4t + 3,98E-6.

Si se ha encontrado raíz t_c que cumpla la ecuación (2.51), se puede decir que se cumple la primera condición de colisión en el intervalo, que los cuatro nodos en algún momento sean coplanarios. Sin embargo, para comprobar que hay colisión, también se debe cumplir la segunda condición. Es decir, en el momento en el que los cuatro nodos son coplanarios el nodo $\mathbf{x_4}$ debe estar dentro del triángulo formado por los otros tres nodos. Si no se realizase esta segunda comprobación, un caso

como el segundo de la Figura 2.25 podría registrarse como colisión, dando lugar a un falso positivo.

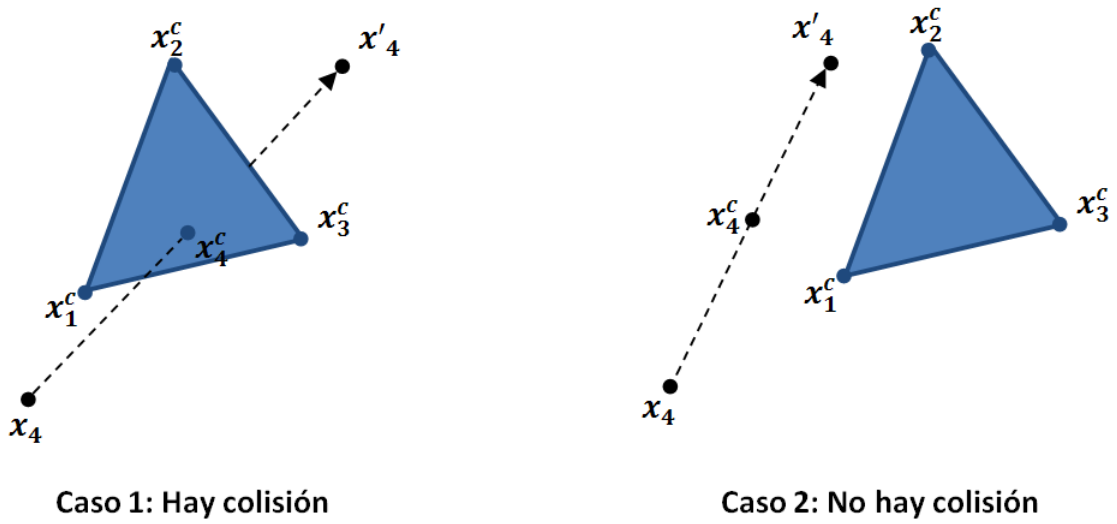


Figura 2.25: En el caso 1 se cumplen las dos condiciones para que haya colisión. Sin embargo, en el caso 2 sólo se cumple la de que los nodos sean coplanarios en un instante de tiempo dentro del intervalo simulado

Una forma sencilla de determinar si $\mathbf{x}^c_4$ se encuentra dentro del triángulo formado por los nodos $\mathbf{x}^c_1$, $\mathbf{x}^c_2$ y $\mathbf{x}^c_3$ consiste en calcular las coordenadas baricéntricas de dicho nodo con respecto al triángulo. Como se explica en el apartado 2.2.2, si el nodo se encuentra dentro del triángulo sus coordenadas baricéntricas deben estar entre 0 y 1. Se pueden calcular las coordenadas baricéntricas a partir de las ecuaciones (2.54), (2.55) y (2.56), adaptaciones directas de las ecuaciones (2.30), (2.31) y (2.32).

$$w_1 = \frac{\begin{vmatrix} \mathbf{x}^c_{13} \cdot \mathbf{x}^c_{43} & \mathbf{x}^c_{13} \cdot \mathbf{x}^c_{23} \\ \mathbf{x}^c_{23} \cdot \mathbf{x}^c_{43} & \mathbf{x}^c_{23} \cdot \mathbf{x}^c_{23} \end{vmatrix}}{\begin{vmatrix} \mathbf{x}^c_{13} \cdot \mathbf{x}^c_{13} & \mathbf{x}^c_{13} \cdot \mathbf{x}^c_{23} \\ \mathbf{x}^c_{23} \cdot \mathbf{x}^c_{13} & \mathbf{x}^c_{23} \cdot \mathbf{x}^c_{23} \end{vmatrix}} \tag{2.54}$$

$$w_2 = \frac{\begin{vmatrix} \mathbf{x}^c_{13} \cdot \mathbf{x}^c_{13} & \mathbf{x}^c_{13} \cdot \mathbf{x}^c_{43} \\ \mathbf{x}^c_{23} \cdot \mathbf{x}^c_{13} & \mathbf{x}^c_{23} \cdot \mathbf{x}^c_{43} \end{vmatrix}}{\begin{vmatrix} \mathbf{x}^c_{13} \cdot \mathbf{x}^c_{13} & \mathbf{x}^c_{13} \cdot \mathbf{x}^c_{23} \\ \mathbf{x}^c_{23} \cdot \mathbf{x}^c_{13} & \mathbf{x}^c_{23} \cdot \mathbf{x}^c_{23} \end{vmatrix}} \tag{2.55}$$

$$w_3 = 1 - w_1 - w_2 \tag{2.56}$$

Dónde:

$$\mathbf{x}^c_{ij} = \mathbf{x}^c_i - \mathbf{x}^c_j \tag{2.57}$$

En el Diagrama 2.9, se muestra el algoritmo completo de detección de colisiones nodo-triángulo que se ha explicado en el presente apartado. Si la pareja nodo-triángulo cumple las condiciones de colisión, se añade a la lista de nodos y triángulos sobre los que se ha de aplicar un impulso inelástico, a fin de evitar la misma.

Se recuerda al lector que este algoritmo se basa en la suposición de que el desplazamiento de los nodos durante el intervalo de tiempo es lineal, a falta de más información. Por ello, la precisión de la detección de colisiones aumenta al disminuir el paso de simulación. Sin embargo, este tipo de suposición se realiza en la mayoría de simulaciones numéricas y, en este caso, tiene un impacto despreciable sobre la simulación, siempre y cuando el paso sea pequeño.

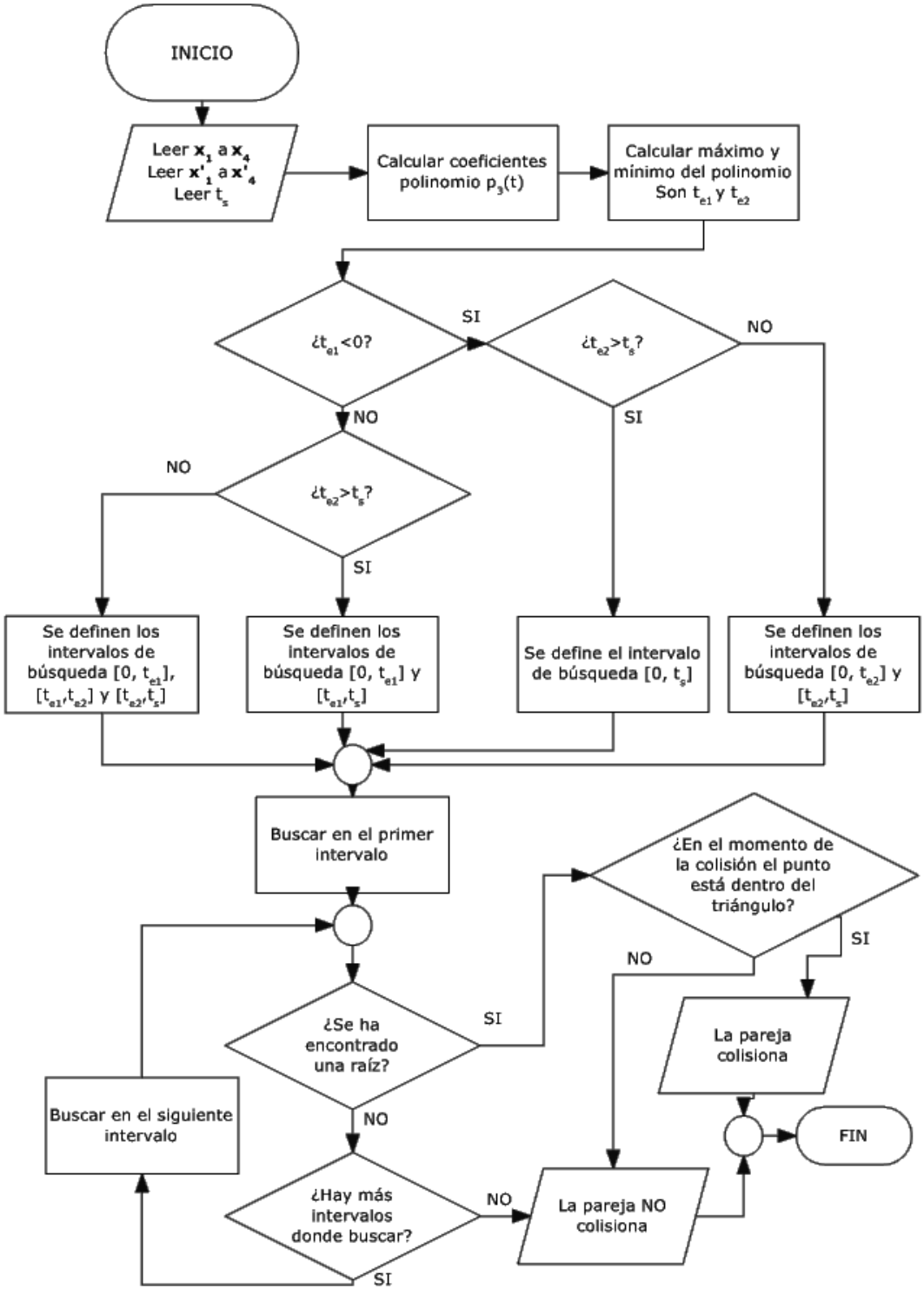


Diagrama 2.9: Algoritmo de determinación de colisión entre un nodo y un triángulo en un intervalo de tiempo. Se parte de las posiciones iniciales, las finales y del intervalo en el que se desplazan los nodos de las unas a las otras.

Detección de colisiones entre aristas

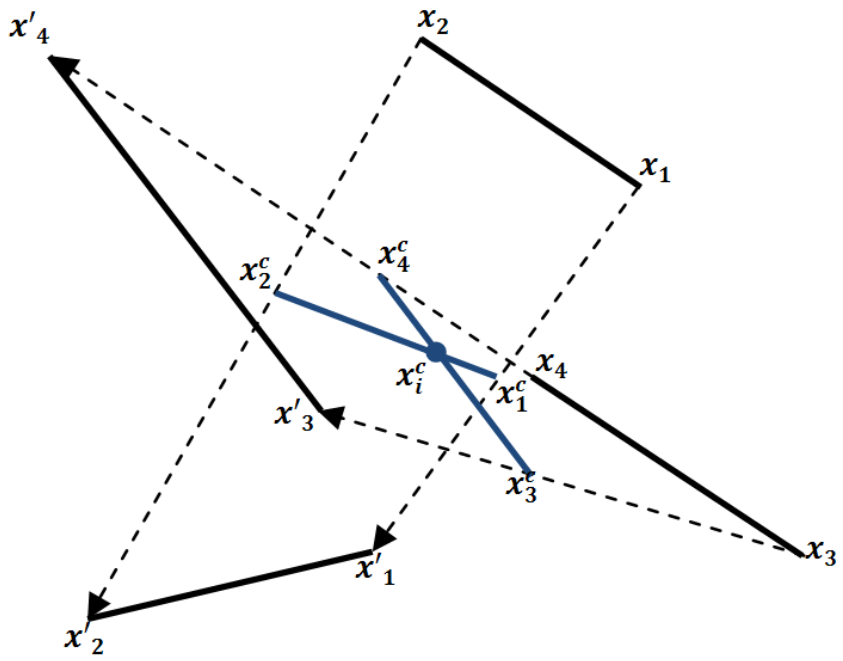


Figura 2.26: Colisión entre dos aristas y nomenclatura empleada

La terminología utilizada para la detección precisa de colisión entre dos aristas es parecida a la utilizada en la detección precisa de colisiones nodo-triángulo, ver la Figura 2.26:

$\mathbf{x_1}$ a $\mathbf{x_4}$ son las posiciones de los nodos de las aristas al inicio del lapso de tiempo simulado.

$\mathbf{x'_1}$ a $\mathbf{x'_4}$ son las posiciones de los nodos de las aristas al final del lapso de tiempo simulado.

$\mathbf{x^c_1}$ a $\mathbf{x^c_4}$ son las posiciones de los nodos de las aristas en el momento de la colisión, en caso de que la haya.

$\mathbf{x^c_i}$ es el punto de intersección de las aristas cuando ocurre la colisión, en caso de que la haya.

También se supone movimiento rectilíneo de los nodos durante el lapso de tiempo simulado en este paso, por lo que se calculan las velocidades de las partículas según la ecuación (2.48).

Para que se determine que hay colisión se han de cumplir dos condiciones:

I. La primera condición es la misma que en el caso de colisiones nodo-triángulo. Ésta es que, en un instante del lapso de tiempo simulado, los cuatro nodos que forman parte de las aristas deben ser coplanarios.

II. La segunda condición es que en el momento en el que los nodos están en el mismo plano, las dos aristas se cruzan, dando lugar a un punto de colisión x^c_i.

A continuación se explica cómo se determina si se cumplen las dos condiciones. La primera condición se puede verificar de la misma forma que en el caso de detección de colisiones nodo-triángulo, ver la ecuación (2.49) y el desarrollo que le sigue. Es decir, se construye un polinomio de tercer grado y se buscan sus raíces aplicando el método de la secante a intervalos de tiempo concretos. Se pueden aplicar las mismas ecuaciones y métodos puesto que se trata del mismo número de nodos, tanto una pareja nodo-triangulo como una arista-arista tienen 4 nodos.

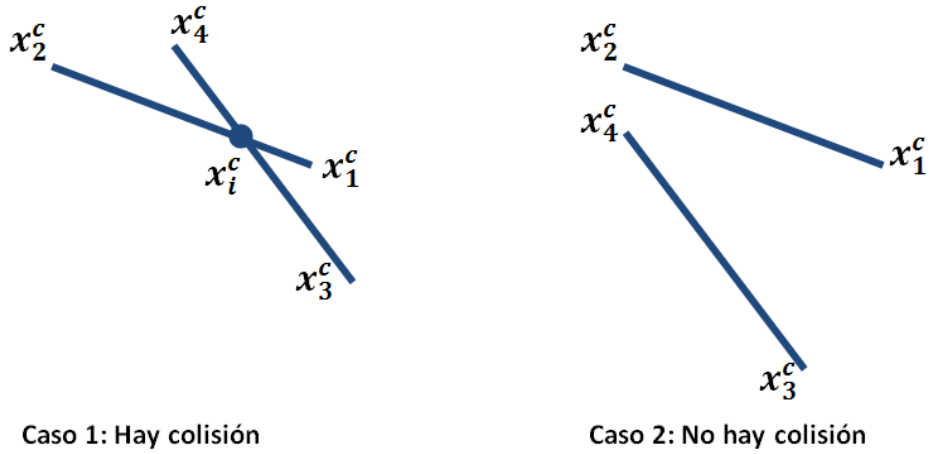


Figura 2.27: Es necesario comprobar que se cruzan las aristas en el momento de la coplanaridad

En caso de que se cumpla la condición de coplanaridad, se pasa a determinar si las aristas se cruzan en el instante en el que ésta ocurre. Se definen *a* y *b* como las distancias, en tanto por uno, respecto a la longitud de cada arista, respecto al origen, $\mathbf{x_1}$ o $\mathbf{x_3}$ en función de la arista, a la que se produce la intersección entre las aristas. Es decir, se deben calcular los parámetros *a* y *b* para que se cumplan las ecuaciones (2.58) y (2.59).

$$\mathbf{x_i^c} = \mathbf{x_1^c} + a\mathbf{x_{21}^c} = (1-a)\mathbf{x_1^c} + a\mathbf{x_2^c} \tag{2.58}$$

$$\mathbf{x_i^c} = \mathbf{x_3^c} + b\mathbf{x_{43}^c} = (1-b)\mathbf{x_3^c} + b\mathbf{x_4^c} \tag{2.59}$$

Igualando las dos ecuaciones y ordenando en forma matricial se llega a la ecuación (2.61).

$$a\mathbf{x_{21}^c} - b\mathbf{x_{43}^c} = \mathbf{x_{31}^c} \tag{2.60}$$

$$\begin{pmatrix}\mathbf{x_{21}^c} & -\mathbf{x_{43}^c}\end{pmatrix}\begin{pmatrix}a\\b\end{pmatrix} = \begin{pmatrix}\mathbf{x_{31}^c}\end{pmatrix} \tag{2.61}$$

La ecuación anterior se debe resolver para *a* y *b*, para ello resulta útil desarrollar los vectores de posición de forma tal que se trabaje con sus componentes, ver ecuación (2.62). Se observa que se trata de un sistema de 3 ecuaciones y 2 incógnitas, lo que podría dar a entender que es un sistema incompatible. Sin embargo, hay una ecuación del sistema que es linealmente dependiente de las otras dos, pues ya se sabe que se da la condición de que las aristas son coplanarias. Por ello, se puede trabajar con cualquiera de los tres subsistemas de dos ecuaciones que se pueden escoger. Para la ecuación (2.63) se ha eliminado la tercera ecuación del sistema. Más adelante, se estudia qué diferencia hay entre escoger un subsistema u otro.

$$\begin{pmatrix}\mathbf{x_{21}^c}|_{\mathbf{x}} & -\mathbf{x_{43}^c}|_{\mathbf{x}}\\ \mathbf{x_{21}^c}|_{\mathbf{y}} & -\mathbf{x_{43}^c}|_{\mathbf{y}}\\ \mathbf{x_{21}^c}|_{\mathbf{z}} & -\mathbf{x_{43}^c}|_{\mathbf{z}}\end{pmatrix}\begin{pmatrix}a\\b\end{pmatrix} = \begin{pmatrix}\mathbf{x_{31}^c}|_{\mathbf{x}}\\ \mathbf{x_{31}^c}|_{\mathbf{y}}\\ \mathbf{x_{31}^c}|_{\mathbf{z}}\end{pmatrix} \tag{2.62}$$

$$\begin{pmatrix} \mathbf{x}_{21}^{c}|_{x} & -\mathbf{x}_{43}^{c}|_{x} \\ \mathbf{x}_{21}^{c}|_{y} & -\mathbf{x}_{43}^{c}|_{y} \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \mathbf{x}_{31}^{c}|_{x} \\ \mathbf{x}_{31}^{c}|_{y} \end{pmatrix} \tag{2.63}$$

El sistema (2.63) ya es un sistema compatible determinado, por lo que se puede resolver, por el método de Cramer, llegando a las soluciones (2.64) y (2.65). Se destaca la facilidad con la que se pueden agrupar los términos finales en los componentes de productos vectoriales para conseguir ecuaciones más compactas.

$$a = \frac{\begin{vmatrix} \mathbf{x}_{31}^{c}|_{x} & -\mathbf{x}_{43}^{c}|_{x} \\ \mathbf{x}_{31}^{c}|_{y} & -\mathbf{x}_{43}^{c}|_{y} \end{vmatrix}}{\begin{vmatrix} \mathbf{x}_{21}^{c}|_{x} & -\mathbf{x}_{43}^{c}|_{x} \\ \mathbf{x}_{21}^{c}|_{y} & -\mathbf{x}_{43}^{c}|_{y} \end{vmatrix}} = \frac{\mathbf{x}_{43}^{c}|_{x}\mathbf{x}_{31}^{c}|_{y} - \mathbf{x}_{31}^{c}|_{x}\mathbf{x}_{43}^{c}|_{y}}{\mathbf{x}_{43}^{c}|_{x}\mathbf{x}_{21}^{c}|_{y} - \mathbf{x}_{21}^{c}|_{x}\mathbf{x}_{43}^{c}|_{y}} = \frac{(\mathbf{x}_{43}^{c} \times \mathbf{x}_{31}^{c})|_{z}}{(\mathbf{x}_{43}^{c} \times \mathbf{x}_{21}^{c})|_{z}} \tag{2.64}$$

$$b = \frac{\begin{vmatrix} \mathbf{x}_{21}^{c}|_{x} & \mathbf{x}_{31}^{c}|_{x} \\ \mathbf{x}_{21}^{c}|_{y} & \mathbf{x}_{31}^{c}|_{y} \end{vmatrix}}{\begin{vmatrix} \mathbf{x}_{21}^{c}|_{x} & -\mathbf{x}_{43}^{c}|_{x} \\ \mathbf{x}_{21}^{c}|_{y} & -\mathbf{x}_{43}^{c}|_{y} \end{vmatrix}} = \frac{\mathbf{x}_{21}^{c}|_{x}\mathbf{x}_{31}^{c}|_{y} - \mathbf{x}_{31}^{c}|_{x}\mathbf{x}_{21}^{c}|_{y}}{\mathbf{x}_{43}^{c}|_{x}\mathbf{x}_{21}^{c}|_{y} - \mathbf{x}_{21}^{c}|_{x}\mathbf{x}_{43}^{c}|_{y}} = \frac{(\mathbf{x}_{21}^{c} \times \mathbf{x}_{31}^{c})|_{z}}{(\mathbf{x}_{43}^{c} \times \mathbf{x}_{21}^{c})|_{z}} \tag{2.65}$$

Se observa que las ecuaciones (2.64) y (2.65) poseen un denominador susceptible de ser nulo o cercano a cero, dando errores de cálculo. Por ello, y recurriendo al hecho de que se puede escoger que subsistema de dos ecuaciones del sistema (2.62), resolveremos aquel sistema de dos ecuaciones que provea un denominador mayor. En las ecuaciones (2.66) y (2.67) se muestran las soluciones a calcular en función del valor de las componentes del producto vectorial $(\mathbf{x}_{43}^{c} \times \mathbf{x}_{21}^{c})$.

$$\begin{aligned} &\text{Si } (\mathbf{x}_{43}^{c} \times \mathbf{x}_{21}^{c})|_{x} \text{ máximo } \Rightarrow a = \frac{(\mathbf{x}_{43}^{c} \times \mathbf{x}_{31}^{c})|_{x}}{(\mathbf{x}_{43}^{c} \times \mathbf{x}_{21}^{c})|_{x}} \\ &\text{Si } (\mathbf{x}_{43}^{c} \times \mathbf{x}_{21}^{c})|_{y} \text{ máximo } \Rightarrow a = \frac{(\mathbf{x}_{43}^{c} \times \mathbf{x}_{31}^{c})|_{y}}{(\mathbf{x}_{43}^{c} \times \mathbf{x}_{21}^{c})|_{y}} \\ &\text{Si } (\mathbf{x}_{43}^{c} \times \mathbf{x}_{21}^{c})|_{z} \text{ máximo } \Rightarrow a = \frac{(\mathbf{x}_{43}^{c} \times \mathbf{x}_{31}^{c})|_{z}}{(\mathbf{x}_{43}^{c} \times \mathbf{x}_{21}^{c})|_{z}} \end{aligned} \tag{2.66}$$

$$\begin{aligned} &\text{Si } (\mathbf{x}_{43}^{c} \times \mathbf{x}_{21}^{c})|_{x} \text{ máximo } \Rightarrow b = \frac{(\mathbf{x}_{21}^{c} \times \mathbf{x}_{31}^{c})|_{x}}{(\mathbf{x}_{43}^{c} \times \mathbf{x}_{21}^{c})|_{x}} \\ &\text{Si } (\mathbf{x}_{43}^{c} \times \mathbf{x}_{21}^{c})|_{y} \text{ máximo } \Rightarrow b = \frac{(\mathbf{x}_{21}^{c} \times \mathbf{x}_{31}^{c})|_{y}}{(\mathbf{x}_{43}^{c} \times \mathbf{x}_{21}^{c})|_{y}} \\ &\text{Si } (\mathbf{x}_{43}^{c} \times \mathbf{x}_{21}^{c})|_{z} \text{ máximo } \Rightarrow b = \frac{(\mathbf{x}_{21}^{c} \times \mathbf{x}_{31}^{c})|_{z}}{(\mathbf{x}_{43}^{c} \times \mathbf{x}_{21}^{c})|_{z}} \end{aligned} \tag{2.67}$$

Una vez calculados los parámetros *a* y *b*, se determina que las aristas se cruzan, y por tanto colisionan, si ambos parámetros están entre 0 y 1.

$$\text{Las aristas se cruzan} \Leftrightarrow 0 \leq a, b \leq 1 \tag{2.68}$$

El punto de intersección de las aristas se puede calcular fácilmente sustituyendo *a* o *b* en las ecuaciones (2.58) o (2.59). En el Diagrama 2.10, se muestra el algoritmo descrito para la detección precisa de colisiones entre aristas. En él, no se explica en detalle la verificación de coplanaridad, pues dicha operación es igual a la descrita en el Diagrama 2.9.

El algoritmo de detección de colisiones detallado en este apartado se implementa en la clase **WorldCollisionDetector**, documentada en el apartado 5.1.12. Ésta llama a la clase **AABBTree** para realizar la detección aproximada de colisiones.

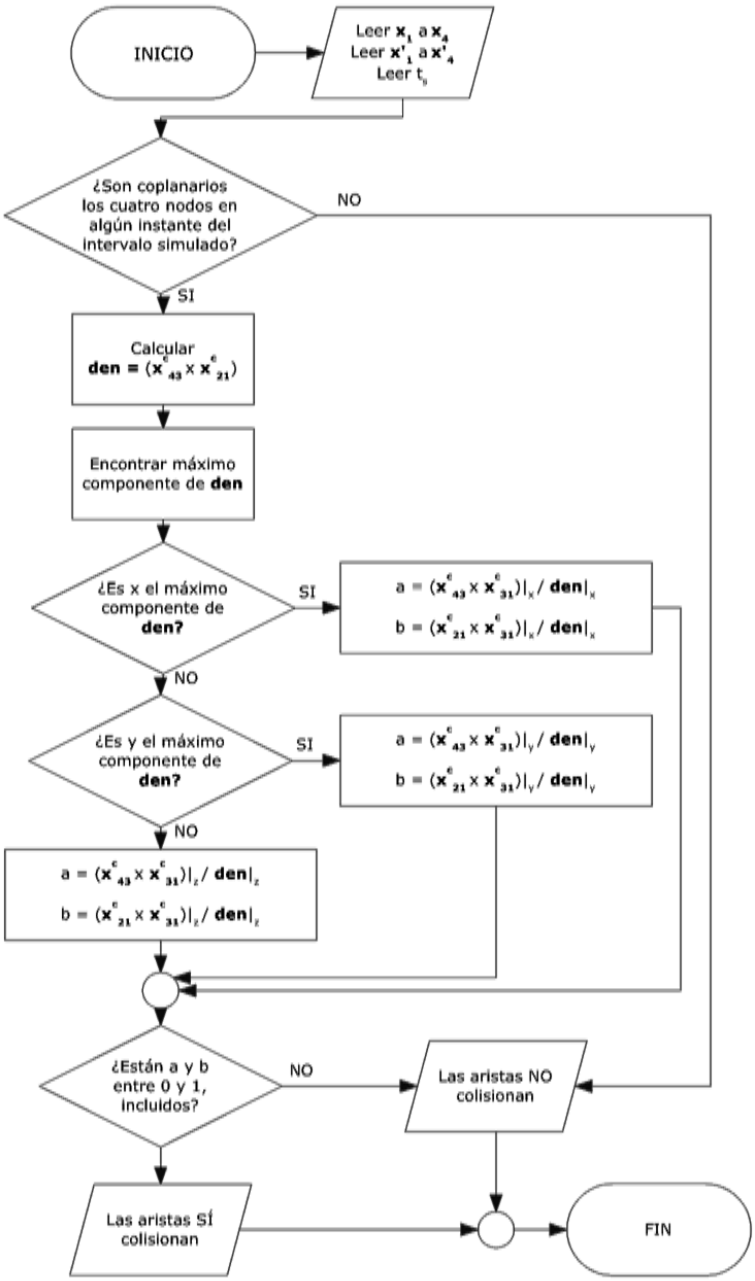


Diagrama 2.10: Algoritmo de detección precisa de colisiones entre aristas

2.2.5 Resolución de contactos y colisiones

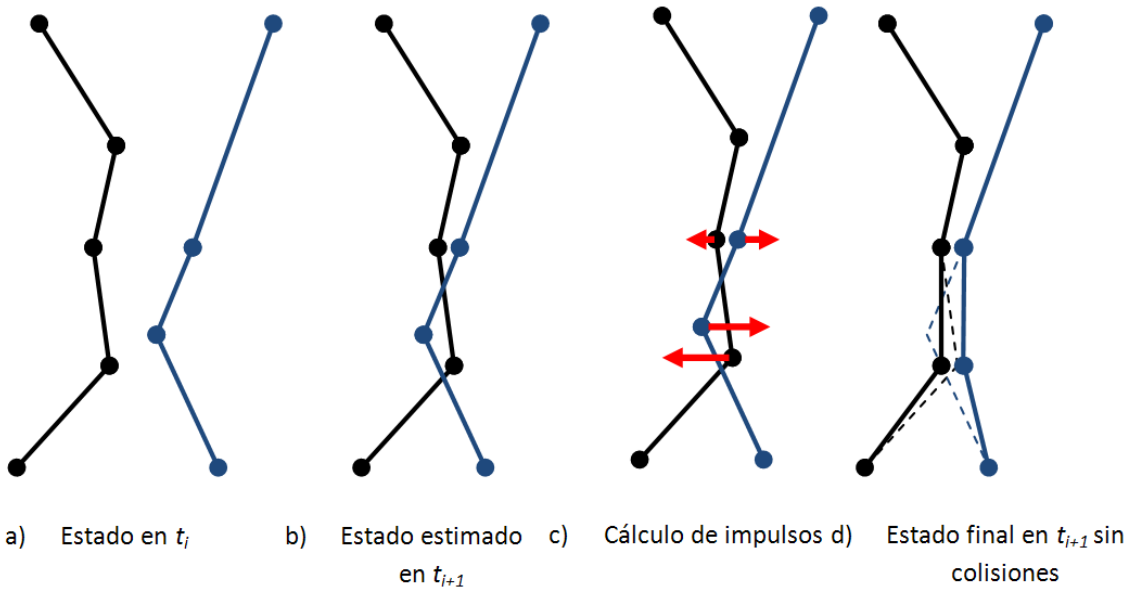


Figura 2.28: Las colisiones y contactos se resuelven aplican impulsos de repulsión

En los apartados 2.2.3 y 2.2.4, se describe la detección de contactos y colisiones de tipo nodo-triángulo y arista-arista. El simulador requiere modificar el estado final de los objetos en función de los contactos y colisiones detectadas, para que evitar que los objetos se atraviesen. Para ello, se aplican *impulsos* a los nodos de los objetos presentes en la escena para separar aquellos elementos que iban a colisionar, o estaban en contacto, en un lapso de tiempo simulado.

La Figura 2.28 muestra gráficamente el proceso de detección y resolución de contactos y colisiones. En ella, se muestran dos mallas con un estado inicial, t_i, libre de colisiones, es fundamental que así sea. Mediante un simulador de tejidos basado en la dinámica interna de los tejidos, se simula la evolución de la ropa hasta un estado t_{i+1}. Este estado es una estimación, al no tener en cuenta las colisiones, y es posible que presente mallas que se solapan con otras. Por ello, en los apartados 2.2.3 y 2.2.4 se presentan algoritmos que permiten detectar estos contactos y colisiones y, en este apartado, se presenta cómo resolverlas. La resolución se basa en la aplicación de impulsos, *c* en la Figura 2.28, que separen las mallas y evite colisiones, llegando a un estado final libre de colisiones, *d* en la Figura 2.28.

Se entiende por impulso a la integral de la fuerza que actúa sobre un cuerpo respecto al tiempo. El impulso produce una variación de velocidad, como se muestra en la ecuación (2.69). Cabe destacar que, como se puede observar en los límites de la integral, en cada paso simulado se toma como referencia el instante en el que se inicia el paso, por lo que el final del paso se corresponde al intervalo simulado t_s.

$$\mathbf{I} = \int_0^{t_s} \mathbf{F}(t)dt = \mathbf{F}'\Delta t = m\Delta\mathbf{v} \qquad (2.69)$$

Como se explica en el apartado 2.2.1, los impulsos modifican la velocidad media de las partículas en el lapso de tiempo simulado. La velocidad media estimada, sin tener en cuenta contactos ni colisiones, se calcula a partir de la ecuación (2.70) (Bridson, Fedkiw, & Anderson, 2002).

$$\tilde{\mathbf{v}}^{\mathbf{n+1/2}} = \frac{\tilde{\mathbf{x}}^{\mathbf{n+1}} - \mathbf{x}^{\mathbf{n}}}{\Delta t} \tag{2.70}$$

Dónde:

$\tilde{\mathbf{v}}^{\mathbf{n+1/2}}$: Velocidad media estimada, únicamente teniendo en cuenta la dinámica interna de la ropa, en el intervalo de tiempo simulado. Éste vector es un vector columna con 3n valores, uno por cada componente de cada nodo presente en la malla.

$\tilde{\mathbf{x}}^{\mathbf{n+1}}$: Posición estimada al final del paso simulado, sin tener en cuenta contactos ni colisiones.

$\mathbf{x}^{\mathbf{n}}$: Posición de la malla al principio del paso simulado, para que el algoritmo funcione correctamente el estado inicial debe estar libre de colisiones.

Δt: Intervalo de tiempo simulado en el paso. No se tienen datos de que ocurre dentro de este intervalo de tiempo, sólo de lo que había antes y de lo que se estima que habrá después. A este parámetro, en los apartados 2.2.3 y 2.2.4 se le denomina t_s.

Hay dos tipos de impulso:

I. Impulsos de contacto:

Son aquellos impulsos que se deben a la proximidad de dos elementos al inicio del paso de simulación. Este tipo de impulsos se aplican entre tejido y objeto rígido o entre tejido y tejido. Se modela el contacto como una fuerza de repulsión entre los elementos debida a la compresión de la fibra. Este tipo de impulso se aplica cuando los elementos están a una distancia menor al grosor del tejido, en el apartado 2.2.3 se explica la detección de contactos. La Figura 2.29 muestra un tejido, en el que la línea continua es la fibra neutra y la discontinua las capas del tejido teniendo en cuenta su grosor, que está en contacto con un objeto rígido. Se observa, que el espesor del tejido en la zona de contacto es menor que el espesor nominal, por lo que el tejido en la zona de contacto está *comprimido*. Debido a esta compresión, se generan fuerzas de repulsión que separan el tejido y el objeto rígido, llegando al estado, más relajado, que se muestra en la Figura 2.30.

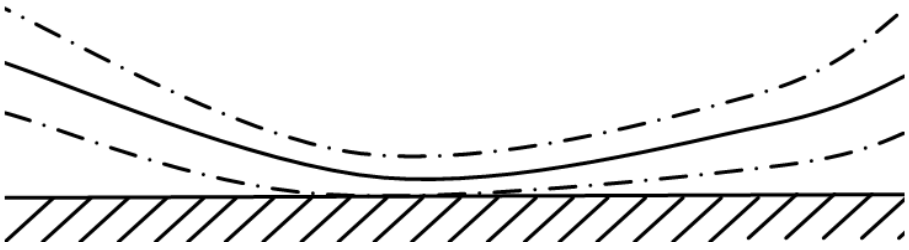


Figura 2.29: Tejido comprimido debido a contacto con objeto rígido

Figura 2.30: El contacto se resuelve y el tejido aparta del objeto rígido recuperando su grosor nominal

Los impulsos de contacto se aplican al principio del paso, pues sólo dependen de la posición inicial de las mallas.

Teóricamente, teniendo en cuenta únicamente las colisiones, sin tener en cuenta los contactos, se podría realizar simulaciones sin que los objetos se atravesaran. Sin embargo, esto no es recomendable por tres razones:

i. La resolución de impulsos de colisión es un método iterativo que puede llegar a ser computacionalmente costoso. Si se le añade a este hecho el que en muchas simulaciones, en las que se tiene un tejido extendido sobre un objeto rígido, hay gran multitud de colisiones a resolver en cada paso, se llega a la conclusión de que cuanto menor sea el número de colisiones a resolver, más rápido será el programa. Los contactos son ideales pues sólo se deben resolver una vez por cada paso simulado y son poco costosos de aplicar. Por ello, interesa tener impulsos de contacto que mantengan los objetos separados para evitar futuras colisiones.

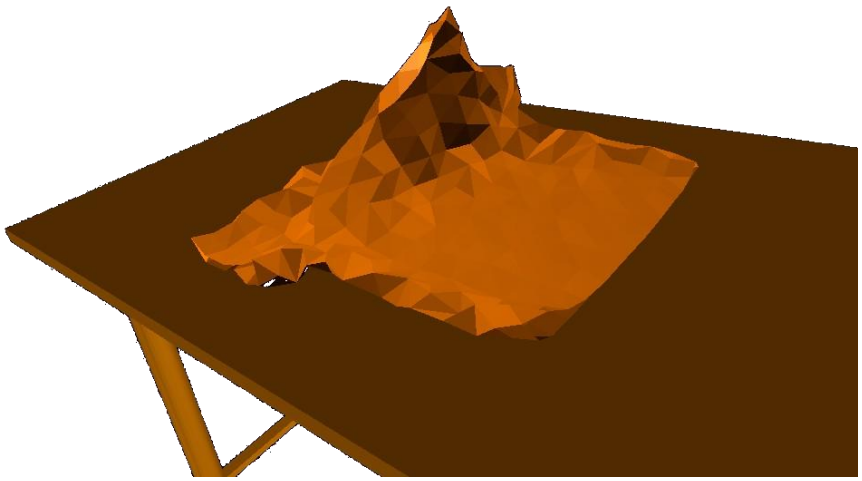


Figura 2.31: Los contactos entre la camiseta y la mesa, mantienen a ambos objetos separados y evitan la resolución de colisiones a cada paso. Imagen tomada del simulador desarrollado *dpoSimulator*.

La Figura 2.31 muestra un caso en el que la aplicación de contactos acelera mucho el cálculo. En caso de que no se aplicasen contactos, en cada paso de la simulación se tendrían que resolver las colisiones que existirían entre la mesa y la camiseta. Los contactos mantienen la camiseta flotando a un milímetro de distancia, correspondiente al grosor del tejido, sobre la superficie de la mesa. Cabe destacar que cuanto menor sea el paso de simulación, mayor será la importancia que cobrarán los contactos, pues se aplicarán más veces, por lo que se deberán resolver menos colisiones.

ii. Las colisiones frenarían el avance de los objetos entre sí, pero podría llegar un momento en el que los elementos estuviesen tan cercanos que por errores de redondeo no se detectasen colisiones. Por lo que los contactos sirven como método de seguridad para evitar futuras colisiones.

iii. Los contactos, independientemente de la forma en que se modelen, son fenómenos reales a tener en cuenta.

Como se explica en el apartado 2.2.1, se aplican todos los impulsos de contacto a la primera estimación de la velocidad media durante el paso para obtener una nueva

estimación que tiene en cuenta los contactos, pero no las colisiones. En el mejor de los casos, la aplicación de los impulsos de contacto habrá subsanado la mayoría de colisiones que habrían ocurrido si no se hubiesen aplicado.

$$\tilde{\mathbf{V}}^{\mathbf{n+1/2}} \xrightarrow{\text{Aplicación impulsos de contacto}} \tilde{\mathbf{V}}_{\mathbf{c}}^{\mathbf{n+1/2}} \tag{2.71}$$

II. Impulsos de colisión:

Los impulsos de colisión se aplican sobre los nodos que forman los elementos que van a colisionar para evitar la colisión. Como se explica en el apartado 2.2.1, los impulsos de colisión se aplican de forma iterativa, mejorando cada vez la estimación de la velocidad media durante el paso, hasta obtener un estado final libre de colisiones. La estimación inicial sobre la que se trabaja es aquella en la que ya se había tenido en cuenta los contactos, ver (2.72).

$$\tilde{\mathbf{V}}_{\mathbf{c}}^{\mathbf{n+1/2}} \xrightarrow{\text{Aplicación iterativa de impulsos de colisión}} \mathbf{V}^{\mathbf{n+1/2}} \tag{2.72}$$

El Diagrama 2.11 muestra cómo se aplican impulsos de colisión iterativamente. En él se ha incluido un contador de iteraciones *i* y un número máximo de iteraciones a realizar, *imax*. En *dpoSimulator*, el simulador desarrollado, cuando se supera el número máximo de iteraciones, se le da la opción al usuario de volver a ejecutar otra tanda de iteraciones o continuar con colisiones. Esta última opción no es recomendable, pues en los pasos subsiguientes los objetos que se han intersectado no se podrían *despegar* debido a las fuerzas de repulsión. Sin embargo, puede ser útil para ver que colisiones no se han podido resolver con la tanda de iteraciones.

Los impulsos se aplican en puntos concretos de los elementos sobre los que actúan. Existen los *puntos de contacto*, que son aquellos puntos más cercanos de dos entidades diferentes que están en contacto, y los *puntos de colisión*, que son aquellos que se intersectan en un momento dado. Por cada contacto o colisión hay dos puntos de contacto o colisión, respectivamente, correspondientes a cada elemento que contacta o colisiona. Puesto que las mallas están formadas por *nodos*, los impulsos que afectan a puntos concretos de la geometría deben *distribuirse* para afectar a la velocidad media de los nodos durante un intervalo de tiempo.

Los impulsos modifican la velocidad media durante el intervalo de tiempo simulado. Por lo que al final se obtiene aquella velocidad media que permite llegar a un estado final libre de colisiones. Este estado final se compone de la *posición* de los nodos y de su *velocidad*. Para que la simulación sea congruente, se tendrá que modificar la velocidad final para que se corresponda con la posición final de los nodos, libre de colisiones.

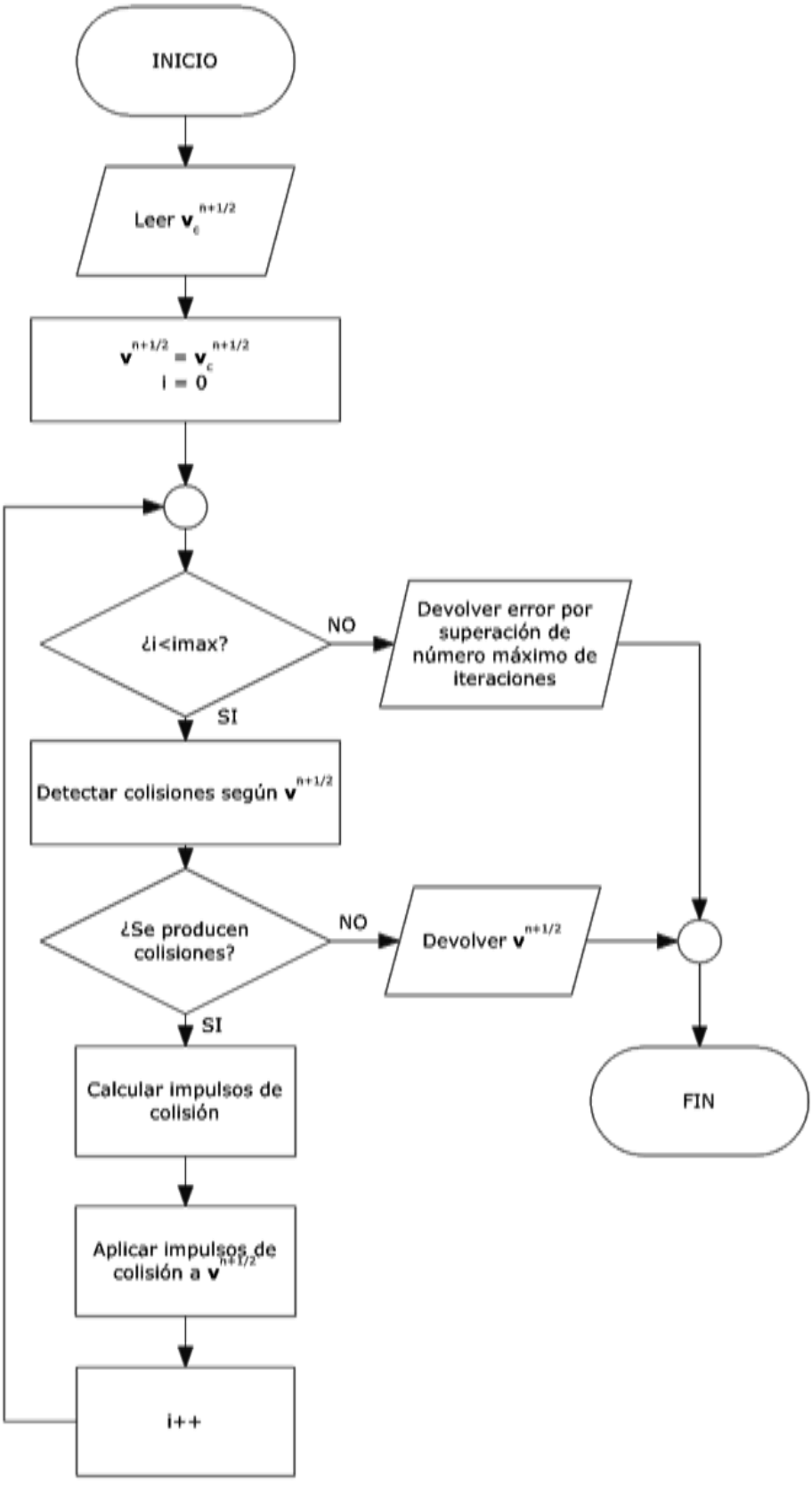


Diagrama 2.11: Algoritmo de aplicación iterativa de impulsos de colisión hasta obtener un estado final libre de interferencias o superar el límite de iteraciones, *imax*.

Modificación de posiciones

Como se ha comentado en el apartado anterior, los impulsos se aplican sobre las velocidades medias de los nodos durante un paso de simulación. La posición de los nodos al final del paso se puede calcular a partir de la ecuación (2.73).

$$\mathbf{x}^{\mathbf{n+1}} = \mathbf{x}^{\mathbf{n}} + \Delta t \mathbf{v}^{\mathbf{n+1/2}} \quad (2.73)$$

En los siguientes apartados se detalla el cálculo de los impulsos y como se aplican a $\mathbf{v}^{\mathbf{n+1/2}}$ para modificar las posiciones finales.

Aplicación de impulsos sobre objetos rígidos y nodos restringidos

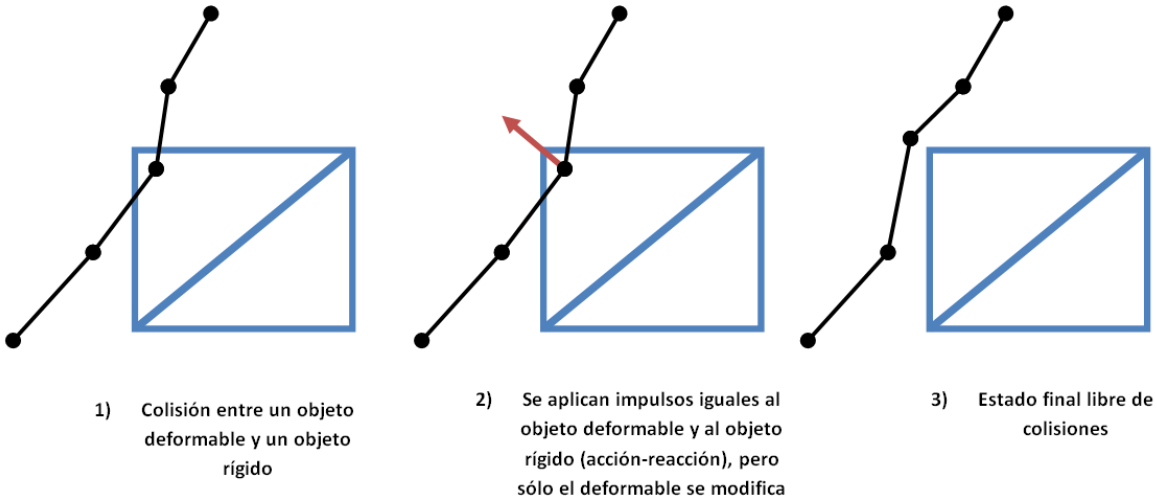


Figura 2.32: Colisión entre un objeto deformable y un objeto rígido, cubo azul.

Antes de describir el cálculo de impulsos y la modificación de las velocidades medias durante un paso, se recuerda que hay dos tipos de objeto considerados por el simulador *dpoSimulator*:

- **Objetos deformables:** Son aquellos que se pueden deformar, cambiar de forma, debido a fuerzas externas, como manipulación de sus nodos o interacciones con otros objetos. Cada nodo de los objetos deformables tiene una masa determinada, que normalmente se calcula igual para todos los nodos de ese objeto. Los objetos deformables, tejidos, son los protagonistas del simulador, pues son aquellos que se simulan con mayor realismo.
- **Objetos rígidos:** Los objetos rígidos no cambian de forma, sólo pueden trasladarse y rotarse. Evidentemente, físicamente ningún objeto es perfectamente rígido. Sin embargo, la rigidez de los objetos que se consideran rígidos es mucho mayor a la de los deformables, por ello se supone rigidez infinita. Además, también se les supone una *masa infinita*. Así, por muy grande que sea el impulso que reciben los objetos rígidos, no se moverán. Se recurre a esta aproximación pues el simulador no tiene la función de simular colisiones entre objetos rígidos, pues el mecanismo no sería el mismo que se explica en este documento. También es comprensible que la mayoría de las simulaciones no se ven alteradas por esta aproximación. Por ejemplo, se puede simular de forma satisfactoria cómo colocar una camisa sobre un maniquí suponiendo que el maniquí, objeto rígido, no se mueve. Puesto que no se simulan colisiones entre objetos rígidos, dos objetos rígidos que vayan a colisionar se atravesarán.

Los objetos deformables se pueden manipular aplicando restricciones de velocidad a los nodos deseados. Al hacerlo, se está diciendo que la posición nodos manipulados durante un paso no se puede ver afectada por colisiones ni contactos. Por ello, también se considera que los nodos manipulados de objetos deformables tienen una masa infinita.

Cálculo de impulsos debidos a una colisión

Las colisiones, cómo se ha explicado en el apartado 2.2.4, se detectan como intersecciones de elementos geométricos en el paso de tiempo simulado bajo la suposición de movimiento rectilíneo, en dicho paso. La detección de colisiones genera una lista de colisiones detectadas, de tipo nodo-triángulo o arista-arista, que se deben tratar y resolver. Para ello se aplican impulsos que separan las dos entidades de tal forma que no se produzca la colisión. En este apartado se describe el cálculo de los impulsos resultantes, uno sobre cada entidad, que se deberán distribuir posteriormente y aplicar a los nodos.

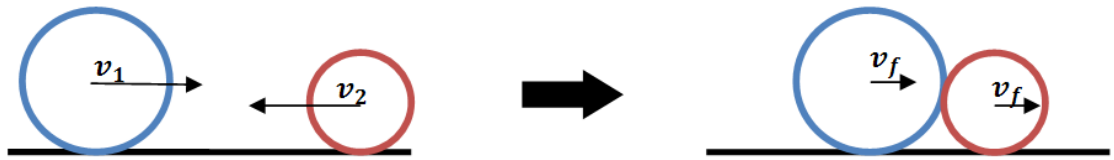


Figura 2.33: Las colisiones entre objetos se modelan como colisiones inelásticas.

Se modelan todas las colisiones como inelásticas. Esto quiere decir que, después de producirse la colisión, la velocidad relativa de las entidades en la dirección normal es nula. Las colisiones entre objetos deformables, tejidos, entre sí o con objetos rígidos se estiman como inelásticas (Bridson, Fedkiw, & Anderson, 2002). Teniendo en cuenta también que no se simula la interacción entre objetos rígidos, pues no forma parte de las funciones del simulador al tener algoritmos de simulación diferentes, se entiende que *todas* las colisiones se modelan como inelásticas.

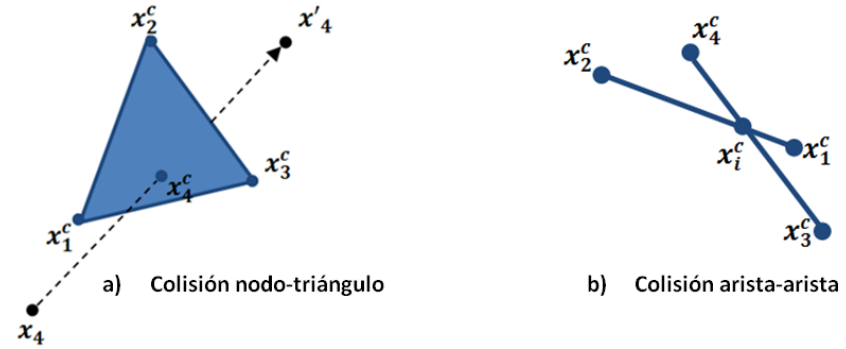


Figura 2.34: Los dos tipos de colisiones se modelan como si fueran monodimensionales, en la dirección normal, entre dos partículas.

Además, las colisiones se modelan con las siguientes características:

i. Se calcula la colisión de un par de entidades como la colisión de dos partículas, para utilizar el modelo sencillo de la Figura 2.22.

    a. En el caso de las colisiones nodo-triángulo se consideran como partículas que colisionan al nodo y a un nodo ficticio que pertenece al triángulo y que tiene, durante todo el paso simulado, las coordenadas baricéntricas del punto donde se

produce la colisión. La primera ecuación en (2.45) permite calcular las coordenadas baricéntricas del punto de colisión, en caso de que no se hayan guardado en el proceso de detección de colisiones. Estas coordenadas se pueden utilizar en la segunda ecuación para calcular la velocidad media del punto de colisión en el paso simulado.

$$\begin{aligned} \mathbf{x_4^c} &= w_1\mathbf{x_1^c} + w_2\mathbf{x_2^c} + w_3\mathbf{x_3^c} \\ \mathbf{v_c^{n+1/2}} &= w_1\mathbf{v_1^{n+1/2}} + w_2\mathbf{v_2^{n+1/2}} + w_3\mathbf{v_3^{n+1/2}} \end{aligned} \tag{2.74}$$

b. Las partículas con colisión inelástica tomadas en consideración en el caso de colisiones entre aristas son dos nodos ficticios que forman parte de las aristas y que tienen las coordenadas baricéntricas del punto de colisión. Por ello, a partir de las coordenadas baricéntricas, *a* y *b*, del punto de colisión respecto cada arista, ver ecuaciones en (2.75), se puede calcular las velocidades de las partículas que colisionan, ecuaciones en (2.76).

$$\begin{cases} \mathbf{x_i^c} = a\mathbf{x_1^c} + (1-a)\mathbf{x_2^c} \\ \mathbf{x_i^c} = b\mathbf{x_3^c} + (1-b)\mathbf{x_4^c} \end{cases} \tag{2.75}$$

$$\begin{cases} \mathbf{v_{c1}^{n+1/2}} = a\mathbf{v_1^{n+1/2}} + (1-a)\mathbf{v_2^{n+1/2}} \\ \mathbf{v_{c2}^{n+1/2}} = b\mathbf{v_3^{n+1/2}} + (1-b)\mathbf{v_4^{n+1/2}} \end{cases} \tag{2.76}$$

ii. Primero se calcula la pareja de impulsos, $\tilde{\mathbf{I}}$, que evitan la colisión al provocar que la velocidad relativa de las partículas en la dirección normal sea nula. Cada impulso *resultante* actuará sobre una partícula diferente y tendrán el mismo módulo y dirección pero sentidos opuestos, por la ley de acción-reacción. Posteriormente, los impulsos resultantes se desglosarán y repartirán entre los nodos.

A partir de las anteriores premisas, muchas de las cuales también son aplicables a la resolución de contactos, se procede a describir la forma de calcular los impulsos resultantes para parejas nodo-triángulo y arista-arista:

a) Cálculo de los impulsos debidos a la colisión de una pareja nodo-triángulo

Se muestra en (2.77) las ecuaciones de la velocidad relativa de las partículas en la dirección normal. Esta velocidad relativa interesa que sea nula, para que la colisión sea inelástica.

$$\begin{aligned} v_N^{n+1/2} &= \left(\mathbf{v_4^{n+1/2}} - \mathbf{v_c^{n+1/2}}\right) \cdot \hat{\mathbf{n}} \overset{\text{def}}{=} 0 \\ v_N^{n+1/2} &= \left(\mathbf{v_4^{n+1/2}} - w_1\mathbf{v_1^{n+1/2}} - w_2\mathbf{v_2^{n+1/2}} - w_3\mathbf{v_3^{n+1/2}}\right) \cdot \hat{\mathbf{n}} \overset{\text{def}}{=} 0 \end{aligned} \tag{2.77}$$

Dónde:

$\mathbf{v_4^{n+1/2}}$: Velocidad de la primera partícula, que no es más que el nodo que colisiona con el triángulo.

$\mathbf{v_c^{n+1/2}}$: Velocidad de la segunda partícula. Es decir, velocidad del punto del triángulo que va a chocar con el nodo. Se calcula a partir de la segunda ecuación en (2.74).

$\hat{\mathbf{n}}$: Dirección normal del triángulo.

También se hace uso de las ecuaciones de distribución de impulsos, demostradas en los siguientes apartados. Éstas permiten modificar las velocidades estimadas, que daban lugar a colisiones, en función de la masa de los nodos, de los impulsos resultantes y de las coordenadas baricéntricas del punto de colisión.

$$\begin{cases} \text{Triángulo} \Rightarrow \mathbf{v_i^{n+1/2}} = \tilde{\mathbf{v}}_\mathbf{i}^{\mathbf{n+1/2}} - \dfrac{w_i}{m_i}\tilde{I}\hat{\mathbf{n}} \Leftarrow \text{i} = 1,2,3 \\ \text{Nodo} \Rightarrow \mathbf{v_4^{n+1/2}} = \tilde{\mathbf{v}}_\mathbf{4}^{\mathbf{n+1/2}} + \dfrac{1}{m_4}\tilde{I}\hat{\mathbf{n}} \end{cases} \tag{2.78}$$

Dónde:

$\mathbf{v_k^{n+1/2}}$: Velocidad media del nodo *k* para evitar la colisión.

$\tilde{\mathbf{v}}_\mathbf{k}^{\mathbf{n+1/2}}$: Velocidad estimada, la de partida, del nodo *k* que provocaba una colisión.

w_i: Coordenada baricéntricas del punto de colisión respecto al nodo *i*.

m_i: Masa del nodo *i*.

$\tilde{I}\tilde{\mathbf{n}}$: Impulso resultante que se aplica sobre el nodo y el triángulo con sentidos opuestos y en la dirección normal. Este impulso es el que se calcula en este apartado.

Para calcular el valor del módulo de los impulsos resultantes se sustituyen las velocidades modificadas, y por ahora desconocidas, de (2.78) en la (2.77). Se puede agrupar los términos de velocidades estimadas obteniendo una velocidad relativa en la dirección normal estimada. El resultado se muestra en la ecuación (2.79). Se observa que la única incógnita es el impulso resultante, por lo que se despeja dicha variable y se obtiene la ecuación final del impulso resultante, ecuación (2.80).

$$\begin{aligned} v_N^{n+\frac{1}{2}} = &\left(\tilde{\mathbf{v}}_\mathbf{4}^{\mathbf{n+\frac{1}{2}}} - w_1\tilde{\mathbf{v}}_\mathbf{1}^{\mathbf{n+\frac{1}{2}}} - w_2\tilde{\mathbf{v}}_\mathbf{2}^{\mathbf{n+\frac{1}{2}}} - w_3\tilde{\mathbf{v}}_\mathbf{3}^{\mathbf{n+\frac{1}{2}}}\right)\cdot\hat{\mathbf{n}} \\ &+\left(\frac{1}{m_4}\tilde{I}\hat{\mathbf{n}} + \frac{w_1^2}{m_1}\tilde{I}\hat{\mathbf{n}} + \frac{w_2^2}{m_2}\tilde{I}\hat{\mathbf{n}} + \frac{w_3^2}{m_3}\tilde{I}\hat{\mathbf{n}}\right)\cdot\hat{\mathbf{n}} = \\ &= \tilde{v}_N^{n+1/2} + \left(\frac{1}{m_4} + \frac{w_1^2}{m_1} + \frac{w_2^2}{m_2} + \frac{w_3^2}{m_3}\right)\tilde{I} \end{aligned} \tag{2.79}$$

$$\tilde{I} = \frac{v_N^{n+1/2} - \tilde{v}_N^{n+1/2}}{\frac{1}{m_4} + \frac{w_1^2}{m_1} + \frac{w_2^2}{m_2} + \frac{w_3^2}{m_3}} \overset{\text{def}}{=} -\frac{\tilde{v}_N^{n+1/2}}{\frac{1}{m_4} + \frac{w_1^2}{m_1} + \frac{w_2^2}{m_2} + \frac{w_3^2}{m_3}} \tag{2.80}$$

La ecuación del impulso resultante es matemáticamente correcta, pero puede causar problemas numéricos cuando uno, o más, de los nodos tiene masa infinita, ya sea porque forma parte de un objeto rígido o porque su movimiento está restringido. Por ello, al programar la ecuación interesa hacerlo sustituyendo cada sumando del denominador por un coeficiente, c_1 a c_4, como muestra la

ecuación (2.81). De esta forma, se puede ir nodo a nodo y anular los coeficientes de todos aquellos nodos de masa infinita. El Diagrama 2.12 muestra el algoritmo completo del cálculo de impulsos resultantes debidos a colisiones entre nodos y triángulos. En él se calculan los coeficientes c_1 a c_4 en función de la masa de los nodos.

$$\tilde{I} = -\frac{\tilde{v}_N^{n+1/2}}{c_1 + c_2 + c_3 + c_4} \tag{2.81}$$

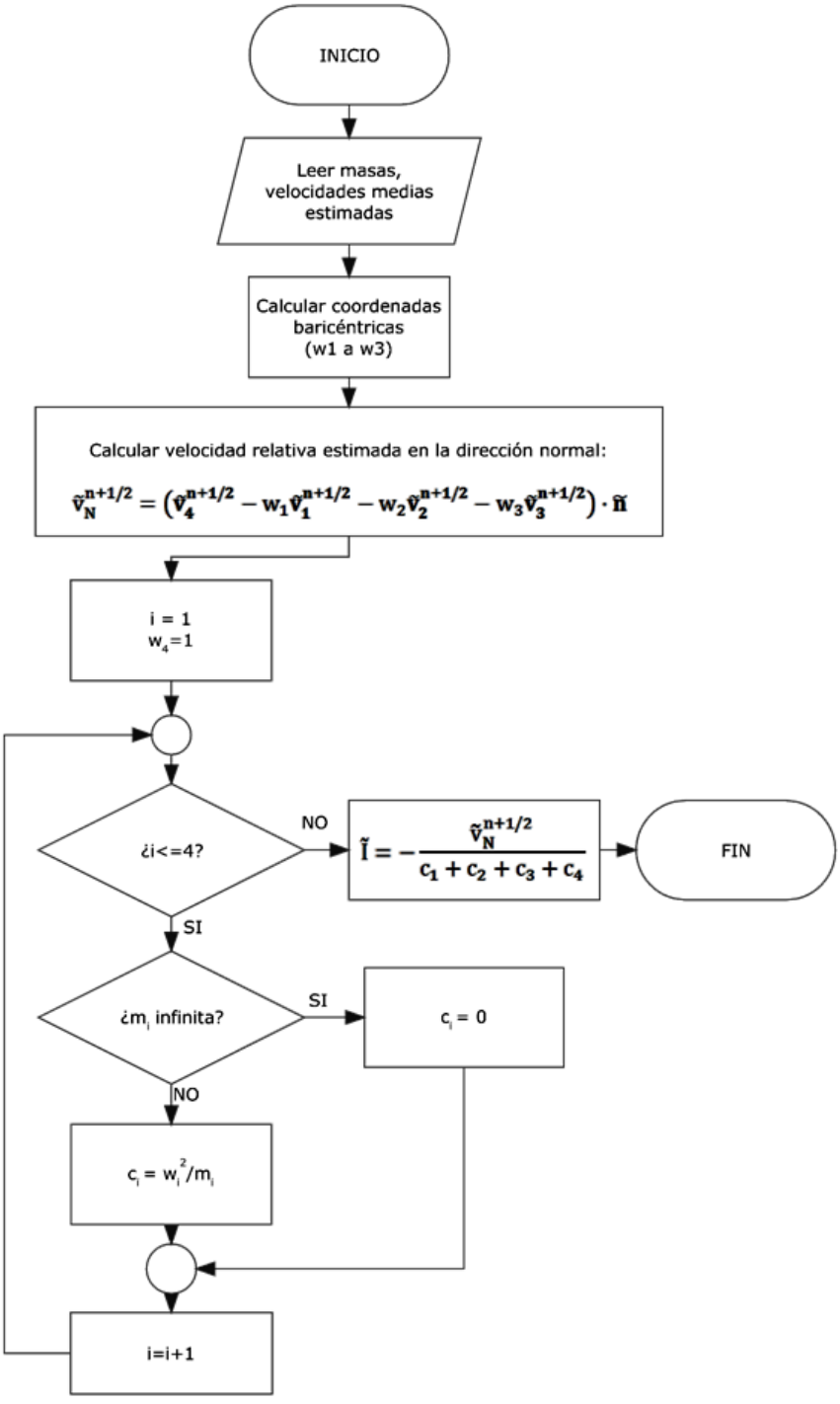


Diagrama 2.12: Algoritmo de cálculo de los impulsos resultantes debido a una colisión entre un nodo y un triángulo, adaptado a posibles objetos rígidos.

b) Cálculo de los impulsos debidos a la colisión de dos aristas:

De forma similar a la anterior, se puede calcula la componente normal de la velocidad relativa entre las dos partículas que van a colisionar y se iguala a cero, debido al choque inelástico. Se sustituyen las ecuaciones de las velocidades de las partículas que van chocar, calculadas en (2.76), en la primera ecuación de (2.82) para dar la segunda ecuación.

$$\begin{aligned} v_N^{n+1/2} &= \left(\mathbf{v_{c1}^{n+1/2}} - \mathbf{v_{c2}^{n+1/2}}\right) \cdot \hat{\mathbf{n}} \stackrel{\text{def}}{=} 0 \\ v_N^{n+1/2} &= \left(a\mathbf{v_1^{n+1/2}} + (1-a)\mathbf{v_2^{n+1/2}} - b\mathbf{v_3^{n+1/2}} - (1-b)\mathbf{v_4^{n+1/2}}\right) \cdot \hat{\mathbf{n}} \stackrel{\text{def}}{=} 0 \end{aligned} \tag{2.82}$$

Dónde:

$\mathbf{v_{c1}^{n+1/2}}$: Velocidad de la partícula que va a colisionar que pertenece a la arista 1.

$\mathbf{v_{c2}^{n+1/2}}$: Velocidad de la partícula que va a colisionar que pertenece a la arista 2.

También se utilizan las ecuaciones de velocidades modificadas, deducidas en los siguientes apartados, para una pareja de aristas. Se han reescrito las ecuaciones en (2.83).

$$\begin{cases} \mathbf{v_1^{n+1/2}} = \tilde{\mathbf{v}}_\mathbf{1}^{\mathbf{n+1/2}} + \dfrac{(1-a)}{m_1}\tilde{I}\hat{\mathbf{n}} \\ \mathbf{v_2^{n+1/2}} = \tilde{\mathbf{v}}_\mathbf{2}^{\mathbf{n+1/2}} + \dfrac{a}{m_2}\tilde{I}\hat{\mathbf{n}} \\ \mathbf{v_3^{n+1/2}} = \tilde{\mathbf{v}}_\mathbf{3}^{\mathbf{n+1/2}} - \dfrac{(1-b)}{m_3}\tilde{I}\hat{\mathbf{n}} \\ \mathbf{v_4^{n+1/2}} = \tilde{\mathbf{v}}_\mathbf{4}^{\mathbf{n+1/2}} - \dfrac{b}{m_4}\tilde{I}\hat{\mathbf{n}} \end{cases} \tag{2.83}$$

Se sustituyen las velocidades modificadas de la ecuación (2.83) en la segunda ecuación de (2.82) y se agrupan los términos correspondientes a las velocidades estimadas para obtener la ecuación final (2.84). De esta ecuación aislamos el impulso resultante, que es aquel que queremos calcular, obteniendo la ecuación final (2.85).

$$\begin{aligned} v_N^{n+\frac{1}{2}} &= \tilde{v}_N^{n+1/2} + \left(\frac{a^2}{m_1}\tilde{I}\hat{\mathbf{n}} + \frac{(1-a)^2}{m_2}\tilde{I}\hat{\mathbf{n}} + \frac{b^2}{m_3}\tilde{I}\hat{\mathbf{n}} + \frac{(1-b)^2}{m_4}\tilde{I}\hat{\mathbf{n}}\right) \cdot \hat{\mathbf{n}} \\ v_N^{n+\frac{1}{2}} &= \tilde{v}_N^{n+1/2} + \left(\frac{a^2}{m_1} + \frac{(1-a)^2}{m_2} + \frac{b^2}{m_3} + \frac{(1-b)^2}{m_4}\right)\tilde{I} \end{aligned} \tag{2.84}$$

$$\tilde{I} = \frac{v_N^{n+1/2} - \tilde{v}_N^{n+1/2}}{\dfrac{a^2}{m_1} + \dfrac{(1-a)^2}{m_2} + \dfrac{b^2}{m_3} + \dfrac{(1-b)^2}{m_4}} \stackrel{\text{def}}{=} -\frac{\tilde{v}_N^{n+1/2}}{\dfrac{a^2}{m_1} + \dfrac{(1-a)^2}{m_2} + \dfrac{b^2}{m_3} + \dfrac{(1-b)^2}{m_4}} \tag{2.85}$$

Igual que en el caso de colisiones nodo-triángulo, resulta útil expresar los sumandos del denominador como coeficientes cuyos valores serán nulos si algún nodo tiene masa infinita, pues forma parte de un objeto rígido o su movimiento está restringido. El algoritmo

completo se muestra en el Diagrama 2.13. El algoritmo de cálculo de impulsos debidos a un contacto se implementa en la clase **WorldCollisionResponser**, apartado 5.1.13.

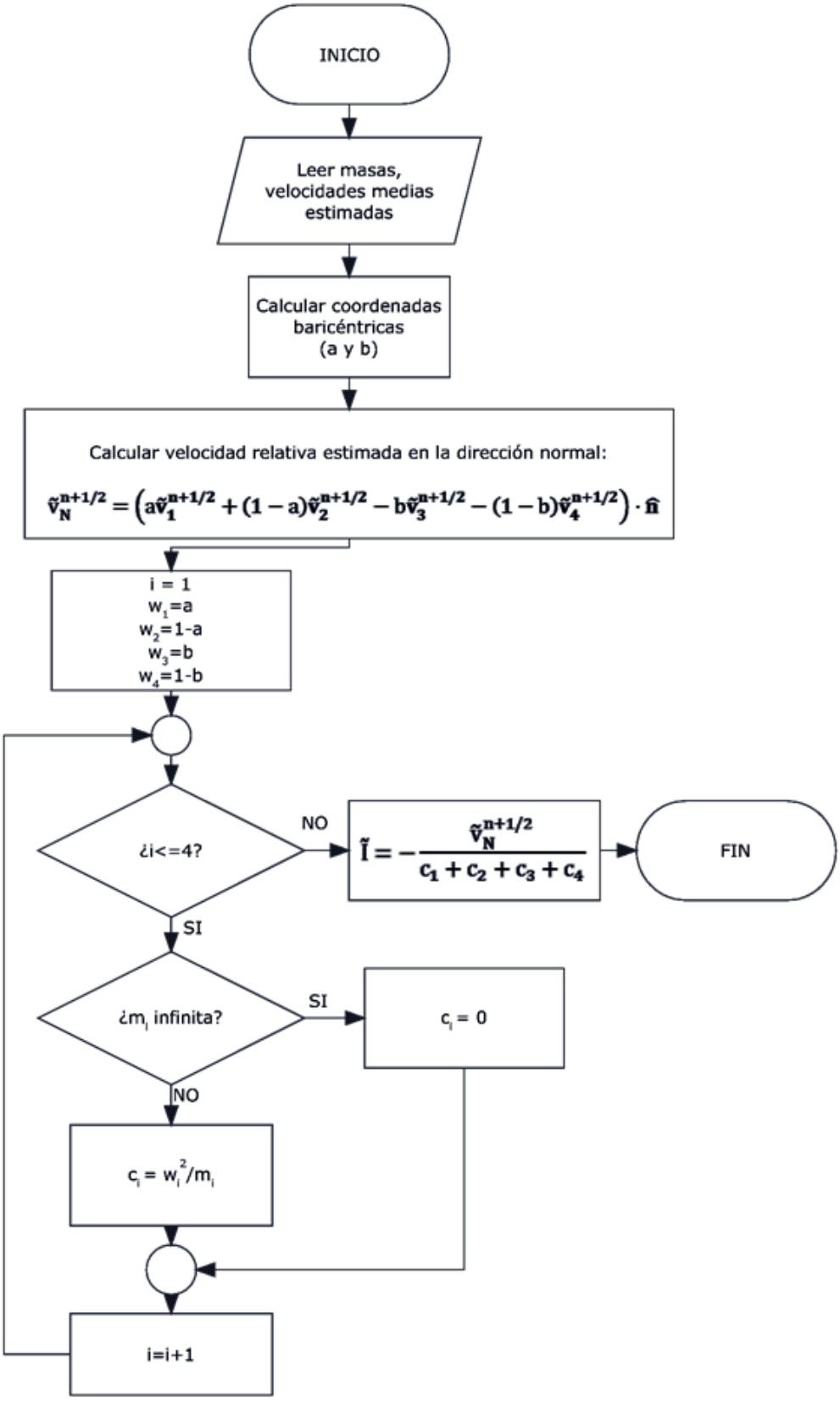


Diagrama 2.13: Algoritmo de cálculo de los impulsos resultantes debido a una colisión entre dos aristas, adaptado a posibles objetos rígidos.

Cálculo de impulsos debidos a un contacto

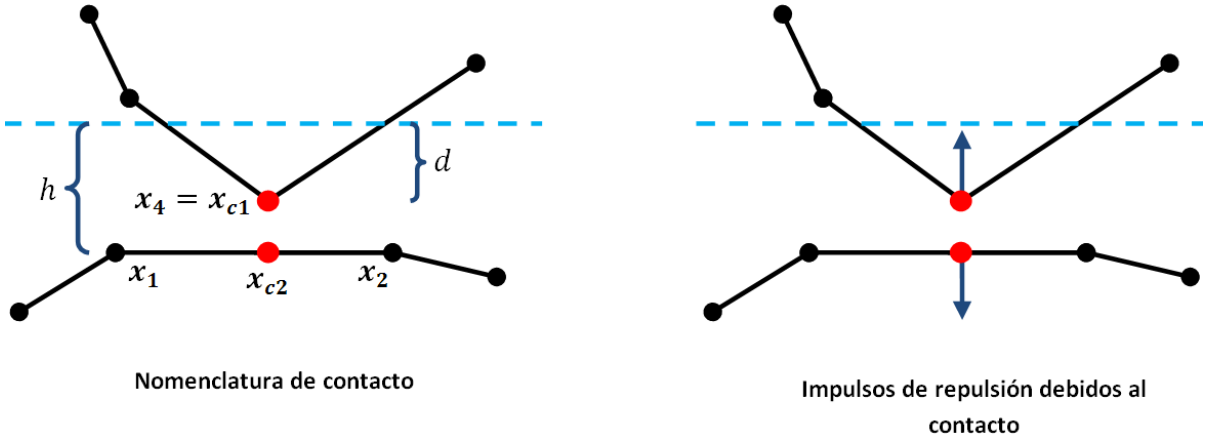


Figura 2.35: El contacto se modela como una repulsión que intenta anular la distancia de interferencia, d.

Se propone un modelo de contacto que cumpla dos requisitos (Bridson, Fedkiw, & Anderson, 2002):

I. Aquellas entidades que estén más cercanas a una distancia *h*, ver el apartado 2.2.3, se verán separadas con una fuerza proporcional a la distancia de interferencia, *d*. Es decir, se modela la fuerza de repulsión debida a un contacto como un muelle que une las dos entidades, de longitud natural *h*, y que al comprimirse las separa. La distancia de solapamiento se puede calcular de forma genérica con la ecuación (2.86).

$$d = h - |(\mathbf{x}_{\mathbf{c1}}^{\mathbf{n}} - \mathbf{x}_{\mathbf{c2}}^{\mathbf{n}}) \cdot \hat{\mathbf{n}}| \qquad (2.86)$$

Dónde $\mathbf{x}_{\mathbf{c1}}^{\mathbf{n}}$ y $\mathbf{x}_{\mathbf{c2}}^{\mathbf{n}}$ son los puntos más cercanos pertenecientes a las dos entidades. La Figura 2.35 muestra el caso de un nodo y un triángulo en contacto. Cuanto mayor es la distancia *d* mayor es la compresión de las fibras del tejido y mayor es la fuerza de repulsión que actúa sobre las dos entidades. Si hay contacto, la distancia de interferencia será positiva. Ésta comprobación se puede realizar para confirmar que la lista de entidades en contacto suministrada por el detector, ver el apartado 2.2.3, es correcta.

II. Se limita la fuerza de repulsión a aquella que *como mucho* separe las entidades un 10% de la distancia de compresión. Esto se hace para evitar artificios debidos a la utilización de muelles demasiado elásticos y para que, los tejidos en contacto, sigan en contacto el tiempo suficiente como para que se note la fricción. Sin esta limitación, no se notaría la fricción en mallas bastas.

Cabe destacar que las ecuaciones deducidas en este capítulo son un caso generalizado de las presentadas en (Bridson, Fedkiw, & Anderson, 2002). En esta fuente, se presentan las ecuaciones suponiendo que *todos los nodos* tienen la misma masa. Esto es útil si se está simulando la interacción de un solo tejido consigo mismo, pero limita mucho el simulador. Por ejemplo, no permite simular el comportamiento de un tejido al entrar en contacto con un objeto rígido, como el caso de una camisa sobre un maniquí. Por ello se ha decidido generalizar las ecuaciones y, por lo tanto, se ha tenido que demostrar su origen con rigor matemático. El lector podrá comprobar que sustituyendo una misma masa para todos los nodos en las ecuaciones de cálculo de impulsos se obtienen las presentadas en el documento de Bridson.

Asimismo, se recuerda al lector que, a diferencia de las colisiones, los contactos se producen *al principio* de un paso de la simulación. Por ello, se observará que los superíndices de los vectores de posición, en las ecuaciones que siguen, son *n*, correspondientes al inicio del paso.

A continuación se demuestra por separado el cálculo de la repulsión modelada como un muelle y su limitación de fuerza para después unir las dos ecuaciones en una sola:

a) Modelado de contactos con un muelle:

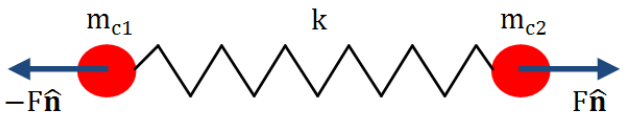


Figura 2.36: Se modela el contacto como un muelle que separa las dos partículas más próximas de las dos entidades.

Partiendo de la ecuación de la fuerza ejercida por un muelle en compresión sobre las dos partículas, ecuación (2.87), se puede calcular el valor de la aceleración de las partículas en la dirección normal, ecuaciones en (2.88).

$$F = kd \tag{2.87}$$

$$\begin{cases} a_{c1N}^{n+1/2} = \mathbf{a_{c1}^{n+1/2}} \cdot \hat{\mathbf{n}} = \dfrac{-F}{m_{c1}} = \dfrac{-kd}{m_{c1}} \\ a_{c2N}^{n+1/2} = \mathbf{a_{c2}^{n+1/2}} \cdot \hat{\mathbf{n}} = \dfrac{F}{m_{c2}} = \dfrac{kd}{m_{c2}} \end{cases} \tag{2.88}$$

Restando ambas ecuaciones se obtiene la aceleración relativa de las dos partículas en la dirección normal, ecuación (2.89).

$$a_N^{n+1/2} = a_{c2N}^{n+1/2} - a_{c1N}^{n+1/2} = F\left(\frac{1}{m_{c1}} + \frac{1}{m_{c2}}\right) \tag{2.89}$$

Se sabe que esta aceleración que se ha introducido en la simulación producirá un cambio en la velocidad media de las partículas durante el paso simulado. Por ello, se puede expresar la variación de la velocidad relativa media, en la dirección normal, de las dos partículas en función de la aceleración, ecuación (2.90).

$$a_N^{n+1/2} = \frac{v_N^{n+1/2} - \tilde{v}_N^{n+1/2}}{\Delta t} \tag{2.90}$$

Aislando la diferencia de velocidades de dicha ecuación y sustituyendo la aceleración de la ecuación (2.89), se obtiene la ecuación final (2.91). Más adelante se explica por qué interesa esta ecuación.

$$v_N^{n+1/2} - \tilde{v}_N^{n+1/2} = a_N \Delta t = \Delta t k d\left(\frac{1}{m_{c1}} + \frac{1}{m_{c2}}\right) \tag{2.91}$$

La constante de rigidez del muelle, *k*, tiene un valor que depende del tejido que se esté simulando. A falta de ensayos empíricos, asignarle el valor de la constante de rigidez de la ropa parece dar buenos resultados, (Bridson, Fedkiw, & Anderson, 2002).

b) Limitación de fuerza de repulsión:

Se quiere limitar la fuerza de separación debida a un contacto para que provoque una separación inferior o igual al 10% de la distancia de interferencia, *d*. Para ello, se busca a continuación aquella variación de la velocidad relativa en la dirección normal, como en la ecuación (2.91), que provoque que la separación sea del 10%. Si la separación es del 10% de *d* se cumplirá que la distancia relativa entre las dos partículas, en la dirección normal, en el paso (n+1) será 0,1d veces superior a la del paso n. Es decir, se cumplirá la ecuación (2.92).

$$\mathbf{x_N^{n+1}} = \mathbf{x_N^n} + 0{,}1\mathbf{d} \tag{2.92}$$

Se puede derivar la diferencia de velocidades relativas medias en la dirección normal para ponerla en función de las posiciones relativas finales, ecuación (2.93). Sustituyendo la ecuación (2.92) en la (2.93), se obtiene la ecuación final de la diferencia de velocidades, (2.94).

$$v_N^{n+1/2} - \tilde{v}_N^{n+1/2} = \frac{x_N^{n+1} - x_N^n}{\Delta t} - \frac{\tilde{x}_N^{n+1} - x_N^n}{\Delta t} = \frac{x_N^{n+1} - \tilde{x}_N^{n+1}}{\Delta t} \tag{2.93}$$

$$\begin{gathered} v_N^{n+1/2} - \tilde{v}_N^{n+1/2} = \frac{x_N^{n+1} - \tilde{x}_N^{n+1}}{\Delta t} = \frac{0{,}1d}{\Delta t} - \frac{\tilde{x}_N^{n+1} - x_N^n}{\Delta t} \\ v_N^{n+1/2} - \tilde{v}_N^{n+1/2} = \frac{0{,}1d}{\Delta t} - \tilde{v}_N^{n+1/2} \end{gathered} \tag{2.94}$$

Para que se cumplan las dos condiciones explicadas anteriormente, la diferencia de velocidades se calculará como la mínima de las ecuaciones (2.91) y (2.94), dando lugar a la ecuación (2.95).

$$v_N^{n+1/2} - \tilde{v}_N^{n+1/2} = \min\left(\Delta t k d\left(\frac{1}{m_{c1}} + \frac{1}{m_{c2}}\right), \frac{0{,}1d}{\Delta t} - \tilde{v}_N^{n+1/2}\right) \tag{2.95}$$

Esta diferencia de velocidades calculada se puede sustituir en las ecuaciones de impulso resultante para obtener los impulsos resultantes debidos a contactos. En el caso de las colisiones, interesaba que la velocidad media modificada fuera nula. La única diferencia para el caso de contactos es que ahora interesa que se cumpla la ecuación (2.95). Sin embargo, se observa que dicha ecuación tiene valores, aún por determinar, que dependerán de si lo que está en contacto es una pareja nodo-triángulo o una pareja de aristas:

a) Cálculo de impulsos resultantes debidos al contacto de un nodo y un triángulo:

Al igual que en el caso de las colisiones la primera partícula en contacto se corresponde al nodo y la segunda al punto del triángulo más cercana al nodo, dato determinado y guardado en la fase de detección. Por ello se puede calcular la distancia de interferencia *d* como se muestra en (2.96). Evidentemente, la masa de la primera partícula se corresponderá a la masa del nodo. La masa de la segunda se puede calcular como promedio de las masas de los nodos que componen los vértices del triángulo. El promedio se pondera según las coordenadas baricéntricas. De esta forma, el nodo más cercano a la

segunda partícula incidirá más sobre su masa. En el caso de que todos los nodos tengan la misma masa, el caso más común, la segunda partícula tendrá la masa de los nodos.

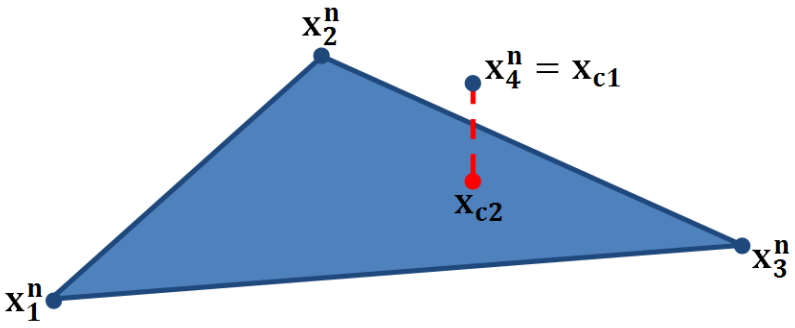


Figura 2.37: Nomenclatura utilizada para un nodo y un triángulo en contacto.

$$\begin{cases} d = h - (\mathbf{x_{c1}^n} - \mathbf{x_{c2}^n}) \cdot \hat{\mathbf{n}} = h - |(\mathbf{x_4^n} - w_1\mathbf{x_1^n} - w_2\mathbf{x_2^n} - w_3\mathbf{x_3^n}) \cdot \hat{\mathbf{n}}| \\ m_{c1} = m_4 \\ m_{c2} = w_1 m_1 + w_2 m_2 + w_3 m_3 \end{cases} \tag{2.96}$$

Se reescribe la ecuación del impulso resultante, debido a una colisión inelástica, calculada en (2.80) y se sustituye la ecuación, (2.95), de diferencia de velocidades para el caso de contactos, para dar lugar a la ecuación final (2.98). También se han sustituido las masas de las partículas en contacto calculadas en (2.96). La ecuación final permite calcular el impulso resultante que se ha de aplicar a un nodo y un triángulo en contacto para separarlos y evitar futuras colisiones.

$$\tilde{I} = \frac{v_N^{n+1/2} - \tilde{v}_N^{n+1/2}}{\frac{1}{m_4} + \frac{w_1^2}{m_1} + \frac{w_2^2}{m_2} + \frac{w_3^2}{m_3}} \tag{2.97}$$

$$\begin{gathered} \tilde{I} = \frac{\min\left(\Delta t k d \left(\frac{1}{m_4} + \frac{1}{w_1 m_1 + w_2 m_2 + w_3 m_3}\right), \frac{0{,}1d}{\Delta t} - \tilde{v}_N^{n+1/2}\right)}{\frac{1}{m_4} + \frac{w_1^2}{m_1} + \frac{w_2^2}{m_2} + \frac{w_3^2}{m_3}} \\ d = h - \left|(\mathbf{x_4^n} - w_1\mathbf{x_1^n} - w_2\mathbf{x_2^n} - w_3\mathbf{x_3^n}) \cdot \hat{\mathbf{n}}\right| \end{gathered} \tag{2.98}$$

b) Cálculo de impulsos resultantes debidos al contacto de dos aristas:

De forma análoga a la anterior, la ecuación (2.99) muestra el valor de la distancia de interferencia y las masas de las partículas en contacto para el caso de dos aristas en contacto. Se recuerda que *a* y *b* son las coordenadas baricéntricas de los puntos en contacto respecto a sus respectivas aristas. Las masas también se calculan como promedio ponderado, según las coordenadas baricéntricas, de las de los vértices.

$$\begin{cases} d = h - (\mathbf{x_{c1}^n} - \mathbf{x_{c2}^n}) \cdot \hat{\mathbf{n}} = h - |(a\mathbf{x_1^n} + (1-a)\mathbf{x_2^n} - b\mathbf{x_3^n} - (1-b)\mathbf{x_4^n}) \cdot \hat{\mathbf{n}}| \\ m_{c1} = am_1 + (1-a)m_2 \\ m_{c2} = bm_3 + (1-b)m_4 \end{cases} \tag{2.99}$$

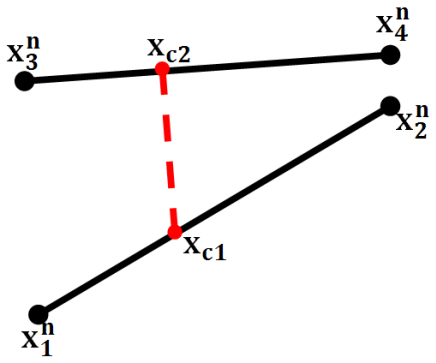


Figura 2.38: Nomenclatura utilizada para dos aristas en contacto.

También se parte de la ecuación de impulso resultante para colisiones inelásticas entre aristas. Se sustituye en ella la diferencia de velocidades, de (2.95) y el valor de las masas, de la (2.99), dando la ecuación final, (2.101), del cálculo de impulsos debidos a un contacto entre dos aristas.

$$\tilde{I} = \frac{v_N^{n+1/2} - \tilde{v}_N^{n+1/2}}{\frac{1}{m_4} + \frac{w_1^2}{m_1} + \frac{w_2^2}{m_2} + \frac{w_3^2}{m_3}} \tag{2.100}$$

$$\tilde{I} = \frac{\min\left(\Delta t k d \left(\frac{1}{a m_1 + (1-a)m_2} + \frac{1}{b m_3 + (1-b)m_4}\right), \frac{0{,}1d}{\Delta t} - \tilde{v}_N^{n+1/2}\right)}{\frac{a^2}{m_1} + \frac{(1-a)^2}{m_2} + \frac{b^2}{m_3} + \frac{(1-b)^2}{m_4}} \tag{2.101}$$

$$d = h - \left|(a\mathbf{x_1^n} + (1-a)\mathbf{x_2^n} - b\mathbf{x_3^n} - (1-b)\mathbf{x_4^n}) \cdot \hat{\mathbf{n}}\right|$$

Distribución de impulsos

Los impulsos calculados en los apartados anteriores, son vectores que actúan sobre cada una de las entidades que forman parte de un contacto o colisión, ya sea nodo, arista o triángulo, en un punto determinado de las mismas y en la dirección normal. Puesto que la información sobre la que se trabaja concierne y transforma a los nodos, se deben distribuir los impulsos a los nodos que forman los elementos. Esta distribución será independiente si el impulso es debido a un contacto o a una colisión, pero sí dependerá del tipo de entidades involucradas.

A continuación se explica cómo se distribuyen los impulsos en función de la pareja de entidades sobre la que actúan:

A. Distribución de impulsos nodo-triángulo:

La distribución de impulsos para una pareja nodo-triángulo consiste en encontrar aquellos impulsos $\mathbf{I_1}$ a $\mathbf{I_4}$ que se pueden aplicar sobre los cuatro nodos provocando una respuesta equivalente a aplicar dos impulsos resultantes y de sentidos opuestos $\tilde{\mathbf{I}}$.

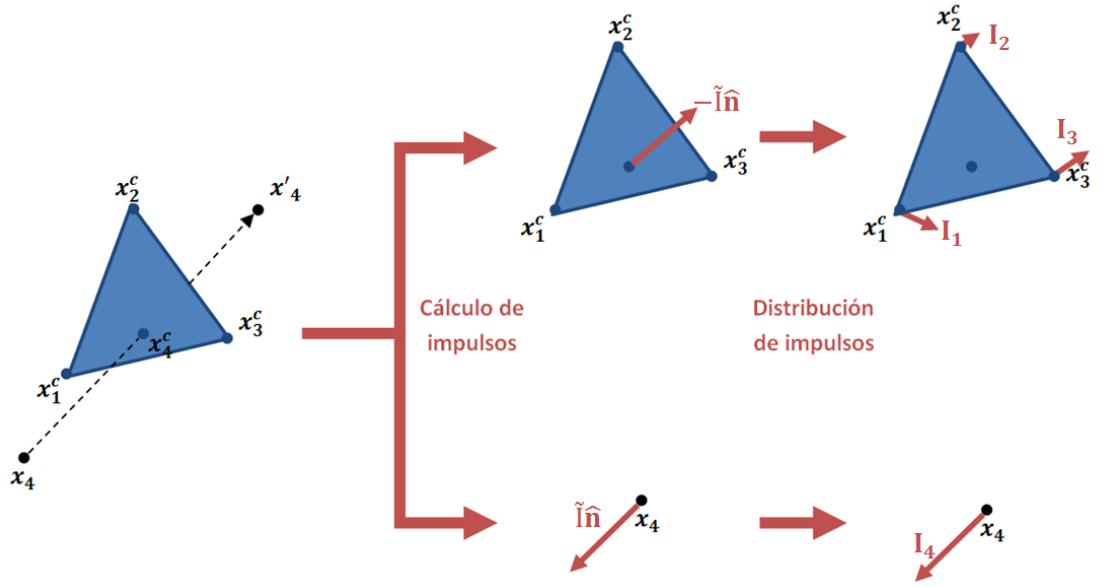


Figura 2.39: Distribución de impulsos nodo-triángulo.

Se debe encontrar una forma de distribuir los impulsos tal que se cumplan las condiciones mostradas en la ecuación (2.102) . La primera condición es que la suma vectorial de impulsos aplicados sobre los nodos del triángulo debe ser igual al impulso total aplicado sobre el triángulo. Esta ecuación se deriva del hecho de que el sumatorio de fuerzas ha de ser la fuerza resultante, teniendo en cuenta que el impulso es la integral de la fuerza a lo largo del tiempo. Se hace notar que el impulso resultante que separa el nodo del triángulo actúa en la dirección normal al triángulo. La segunda condición es que el sumatorio de los momentos debidos a los impulsos aplicados en los nodos del triángulo, respecto al punto dónde se aplica el impulso resultante, debe ser nulo. Con estas dos condiciones, se asegura que al sustituir el impulso resultante por impulsos aplicados en los nodos del triángulo no se introducen ni fuerzas ni momentos adicionales. La tercera condición es evidente si se tiene en cuenta que el impulso que actúa sobre un nodo no se distribuye.

$$\begin{cases} \sum_{i=1}^{3} \mathbf{I_i} = -\tilde{I}\hat{\mathbf{n}} \\ \sum_{i=1}^{3} \mathbf{M_{i|c}} = 0 \\ \mathbf{I_4} = \tilde{I}\hat{\mathbf{n}} \end{cases} \tag{2.102}$$

Dónde:

$\mathbf{I_1}$ a $\mathbf{I_3}$: Impulsos aplicados sobre los nodos del triángulo.

$\mathbf{I_4}$: Impulso aplicado sobre el nodo.

$\tilde{I}$: Módulo del impulso resultante que se aplica sobre el triángulo y el nodo en la misma dirección, normal al triángulo, y sentidos opuestos.

$\mathbf{M_{1|c}}$ a $\mathbf{M_{3|c}}$: Momentos debidos a las fuerzas, derivadas de los impulsos, aplicadas sobre los nodos del triángulo respecto al punto de aplicación del impulso resultante.

Las ecuaciones finales de las velocidades medias modificadas por el impulso resultante sobre cada entidad se muestran en (2.103). Se observa que se trata de una ponderación del impulso resultante en función de las coordenadas baricéntricas y que el efecto del impulso modifica la última estimación de la velocidad media durante el paso actual, $\tilde{\mathbf{v}}_{\mathbf{i}}^{\mathbf{n+1/2}}$, para dar una nueva estimación, $\mathbf{v}_{\mathbf{i}}^{\mathbf{n+1/2}}$. Con estas ecuaciones se puede saber directamente como se aplica una pareja de impulsos resultantes sobre las velocidades medias de los nodos de una pareja nodo-triángulo.

$$\begin{cases} \mathbf{v}_{\mathbf{1}}^{\mathbf{n+1/2}} = \tilde{\mathbf{v}}_{\mathbf{1}}^{\mathbf{n+1/2}} - \dfrac{w_1}{m_1}\tilde{I}\hat{\mathbf{n}} \\ \mathbf{v}_{\mathbf{2}}^{\mathbf{n+1/2}} = \tilde{\mathbf{v}}_{\mathbf{2}}^{\mathbf{n+1/2}} - \dfrac{w_2}{m_2}\tilde{I}\hat{\mathbf{n}} \\ \mathbf{v}_{\mathbf{3}}^{\mathbf{n+1/2}} = \tilde{\mathbf{v}}_{\mathbf{3}}^{\mathbf{n+1/2}} - \dfrac{w_3}{m_3}\tilde{I}\hat{\mathbf{n}} \\ \mathbf{v}_{\mathbf{4}}^{\mathbf{n+1/2}} = \tilde{\mathbf{v}}_{\mathbf{4}}^{\mathbf{n+1/2}} + \dfrac{1}{m_4}\tilde{I}\hat{\mathbf{n}} \end{cases} \tag{2.103}$$

A continuación se comprueba que las ecuaciones mostradas en (2.103) cumplen con las condiciones mostradas en (2.102). Para ello, se parte de la definición de los impulsos que se aplican sobre los nodos. Dichos impulsos producen una variación del momento lineal de los nodos. En la ecuación (2.104) se muestran los valores de los impulsos distribuidos. El lector podrá comprobar que un reordenamiento de los términos llevan a las velocidades medias modificadas de (2.103).

$$\begin{cases} \mathbf{I_1} = m_1\left(\mathbf{v}_{\mathbf{1}}^{\mathbf{n+1/2}} - \tilde{\mathbf{v}}_{\mathbf{1}}^{\mathbf{n+1/2}}\right) = -w_1\tilde{I}\hat{\mathbf{n}} \\ \mathbf{I_2} = m_2\left(\mathbf{v}_{\mathbf{2}}^{\mathbf{n+1/2}} - \tilde{\mathbf{v}}_{\mathbf{2}}^{\mathbf{n+1/2}}\right) = -w_2\tilde{I}\hat{\mathbf{n}} \\ \mathbf{I_3} = m_3\left(\mathbf{v}_{\mathbf{3}}^{\mathbf{n+1/2}} - \tilde{\mathbf{v}}_{\mathbf{3}}^{\mathbf{n+1/2}}\right) = -w_3\tilde{I}\hat{\mathbf{n}} \\ \mathbf{I_4} = m_4\left(\mathbf{v}_{\mathbf{4}}^{\mathbf{n+1/2}} - \tilde{\mathbf{v}}_{\mathbf{4}}^{\mathbf{n+1/2}}\right) = \tilde{I}\hat{\mathbf{n}} \end{cases} \tag{2.104}$$

Se comprueba que estos impulsos cumplen (2.102):

- El sumatorio de los impulsos aplicados sobre el triángulo claramente da como resultado el impulso resultante, teniendo en cuenta que la suma de coordenadas baricéntricas del punto de contacto o colisión es igual a la unidad, ver ecuación (2.105).

$$\sum_{i=1}^{3}\mathbf{I_i} = -(w_1 + w_2 + w_3)\tilde{I}\hat{\mathbf{n}} = -\tilde{I}\hat{\mathbf{n}} \Leftarrow 1 = w_1 + w_2 + w_3 \tag{2.105}$$

- También se comprueba, en (2.106), que el sumatorio de momentos es nulo.

$$\sum_{i=1}^{3}\mathbf{M_{i|c}} = \sum_{i=1}^{3}\left[(\mathbf{x_i^c} - \mathbf{x_4^c}) \times \frac{\mathbf{I_i}}{\Delta t}\right] = \frac{\sum_{i=1}^{3}[(\mathbf{x_i^c} - \mathbf{x_4^c}) \times (-w_i \tilde{I}\hat{\mathbf{n}})]}{\Delta t} = \\ = \frac{\left(\sum_{i=1}^{3}(w_i \mathbf{x_i^c}) - \sum_{i=1}^{3} w_i \mathbf{x_4^c}\right) \times (-\tilde{I}\hat{\mathbf{n}})}{\Delta t} = \frac{(\mathbf{x_4^c} - \mathbf{x_4^c}) \times (-\tilde{I}\hat{\mathbf{n}})}{\Delta t} = 0 \tag{2.106}$$

B. Distribución de impulsos arista-arista:

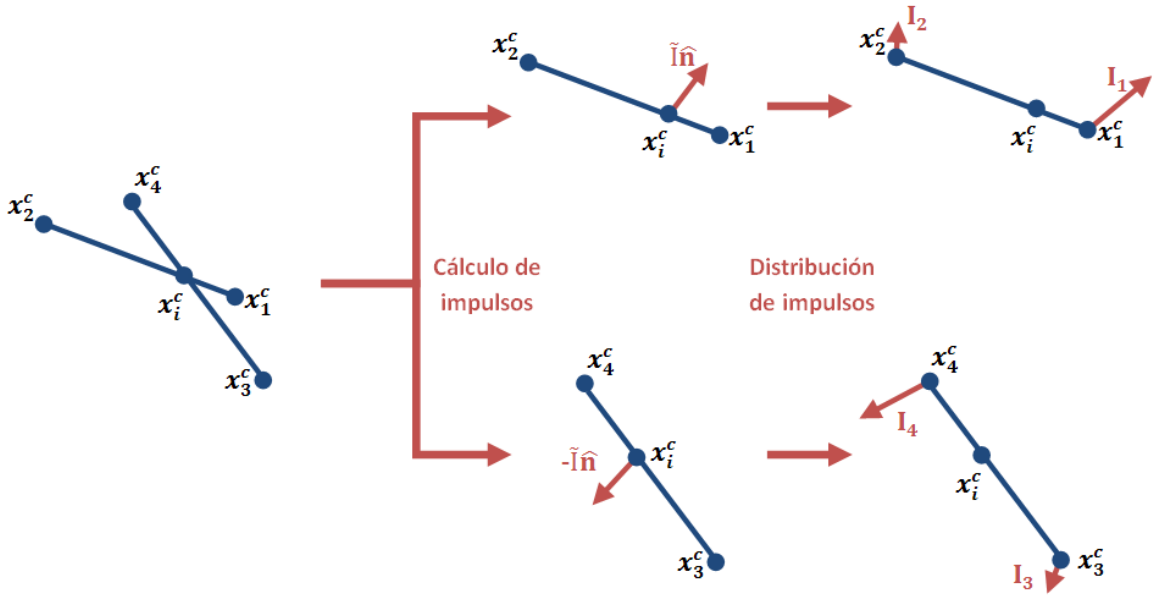


Figura 2.40: Distribución de impulsos arista-arista.

La distribución de impulsos debidos a contactos y colisiones entre dos aristas se puede realizar de forma análoga a la distribución de impulsos nodo-triángulo. En este caso, hay dos impulsos resultantes $\tilde{\mathbf{I}}$ de signos opuestos que se aplican sobre las aristas. Cada impulso resultante se distribuye a los dos nodos que componen la arista, desglosándose en dos impulsos. Los impulsos aplicados sobre los nodos deben cumplir las condiciones mostradas en (2.107). Los impulsos desglosados deben producir el mismo impulso resultante y el mismo momento que el original.

$$\begin{cases} \sum_{k=1}^{2}\mathbf{I_k} = \tilde{I}\hat{\mathbf{n}} \\ \sum_{k=1}^{2}\mathbf{M_{k|i}} = 0 \\ \sum_{k=3}^{4}\mathbf{I_k} = -\tilde{I}\hat{\mathbf{n}} \\ \sum_{k=3}^{4}\mathbf{M_{k|i}} = 0 \end{cases} \tag{2.107}$$

Las ecuaciones en (2.108) permiten calcular la velocidad media de los nodos modificada por los impulsos resultantes. Los valores de los impulsos desglosados se muestran en (2.109). Se destaca que las ecuaciones en (2.109) reordenadas dan lugar a las ecuaciones en (2.108).

$$\begin{cases} \mathbf{v_1^{n+1/2}} = \tilde{\mathbf{v}}_\mathbf{1}^{\mathbf{n+1/2}} + \dfrac{(1-a)}{m_1}\tilde{I}\hat{\mathbf{n}} \\ \mathbf{v_2^{n+1/2}} = \tilde{\mathbf{v}}_\mathbf{2}^{\mathbf{n+1/2}} + \dfrac{a}{m_2}\tilde{I}\hat{\mathbf{n}} \\ \mathbf{v_3^{n+1/2}} = \tilde{\mathbf{v}}_\mathbf{3}^{\mathbf{n+1/2}} - \dfrac{(1-b)}{m_3}\tilde{I}\hat{\mathbf{n}} \\ \mathbf{v_4^{n+1/2}} = \tilde{\mathbf{v}}_\mathbf{4}^{\mathbf{n+1/2}} - \dfrac{b}{m_4}\tilde{I}\hat{\mathbf{n}} \end{cases} \tag{2.108}$$

$$\begin{cases} \mathbf{I_1} = m_1\left(\mathbf{v_1^{n+1/2}} - \tilde{\mathbf{v}}_\mathbf{1}^{\mathbf{n+1/2}}\right) = (1-a)\tilde{I}\hat{\mathbf{n}} \\ \mathbf{I_2} = m_2\left(\mathbf{v_2^{n+1/2}} - \tilde{\mathbf{v}}_\mathbf{2}^{\mathbf{n+1/2}}\right) = a\tilde{I}\hat{\mathbf{n}} \\ \mathbf{I_3} = m_3\left(\mathbf{v_3^{n+1/2}} - \tilde{\mathbf{v}}_\mathbf{3}^{\mathbf{n+1/2}}\right) = -(1-b)\tilde{I}\hat{\mathbf{n}} \\ \mathbf{I_4} = m_4\left(\mathbf{v_4^{n+1/2}} - \tilde{\mathbf{v}}_\mathbf{4}^{\mathbf{n+1/2}}\right) = -b\tilde{I}\hat{\mathbf{n}} \end{cases} \tag{2.109}$$

Se comprueba a continuación que el desglose de impulsos en (2.109) cumple con las condiciones en (2.107):

- Se comprueba que el sumatorio de los impulsos distribuidos equivale a los dos impulsos resultantes, ver las ecuaciones en (2.110).

$$\begin{cases} \displaystyle\sum_{\mathrm{k}=1}^{2} \mathbf{I_k} = \big((1-a)+a\big)\tilde{I}\hat{\mathbf{n}} = \tilde{I}\hat{\mathbf{n}} \\ \displaystyle\sum_{\mathrm{k}=3}^{4} \mathbf{I_k} = -\big((1-b)+b\big)\tilde{I}\hat{\mathbf{n}} = -\tilde{I}\hat{\mathbf{n}} \end{cases} \tag{2.110}$$

- Las sumas de los momentos debidos a las fuerzas, derivadas de los momentos, aplicadas en los nodos, respecto al punto de aplicación de los impulsos resultantes, son nulas. Estas comprobaciones se realizan en (2.111).

$$\begin{cases} \displaystyle\sum_{\mathrm{k}=1}^{2} \mathbf{M_{k|i}} = \sum_{\mathrm{k}=1}^{2}\left[(\mathbf{x_k^c} - \mathbf{x_i^c}) \times \frac{\mathbf{I_k}}{\Delta t}\right] = \frac{(\mathbf{x_1^c} - \mathbf{x_i^c}) \times [(1-a)\tilde{I}\hat{\mathbf{n}}] + (\mathbf{x_2^c} - \mathbf{x_i^c}) \times (a\tilde{I}\hat{\mathbf{n}})}{\Delta t} = \\ \qquad = \dfrac{((1-a)\mathbf{x_1^c} + a\mathbf{x_2^c} - \mathbf{x_i^c}) \times (\tilde{I}\hat{\mathbf{n}})}{\Delta t} = \dfrac{(\mathbf{x_i^c} - \mathbf{x_i^c}) \times (\tilde{I}\hat{\mathbf{n}})}{\Delta t} = 0 \\ \displaystyle\sum_{\mathrm{k}=3}^{4} \mathbf{M_{k|i}} = \sum_{\mathrm{k}=3}^{4}\left[(\mathbf{x_k^c} - \mathbf{x_i^c}) \times \frac{\mathbf{I_k}}{\Delta t}\right] = \frac{(\mathbf{x_3^c} - \mathbf{x_i^c}) \times [(1-b)\tilde{I}\hat{\mathbf{n}}] + (\mathbf{x_4^c} - \mathbf{x_i^c}) \times (b\tilde{I}\hat{\mathbf{n}})}{\Delta \mathrm{t}} = \\ \qquad = \dfrac{((1-b)\mathbf{x_3^c} + b\mathbf{x_4^c} - \mathbf{x_i^c}) \times (\tilde{I}\hat{\mathbf{n}})}{\Delta t} = \dfrac{(\mathbf{x_i^c} - \mathbf{x_i^c}) \times (\tilde{I}\hat{\mathbf{n}})}{\Delta t} = 0 \end{cases} \tag{2.111}$$

Actualización de velocidades

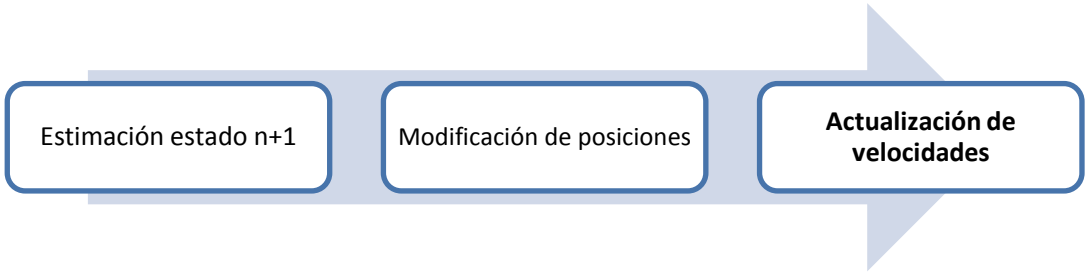


Diagrama 2.14: La actualización de velocidades se ejecuta después de la modificación de posiciones para asegurar que las velocidades son congruentes.

Puesto que se han modificado las posiciones de los nodos, para que no se produzcan colisiones, se deben actualizar las velocidades de los nodos para que sean congruentes, con las modificaciones realizadas. Si esto no se hiciese cuando un trapo en caída libre chocase con una mesa, se evitaría la colisión pero los nodos retendrían la velocidad de caída libre e intentarían continuamente atravesar la mesa. La modificación de posiciones proporciona un conjunto de posiciones $\mathbf{x}^{\mathbf{n+1}}$ que, probablemente, esté libre de interferencias entre objetos.

El método de actualización de velocidades, explicado en este apartado, es una variación del explicado en (Bridson, Fedkiw, & Anderson, 2002), para que sea más preciso. Se parte de la definición de aceleración como derivada de la velocidad, en continuo, y se aproxima a la variación de velocidades a lo largo del paso partido por la duración del mismo, ecuación (2.112).

$$\mathbf{a}\left(t+\frac{\Delta t}{2}\right)=\frac{\partial \mathbf{v}\left(t+\frac{\Delta t}{2}\right)}{\partial t}\approx\frac{\mathbf{v}^{\mathbf{n+1}}-\mathbf{v}^{\mathbf{n}}}{\Delta t} \tag{2.112}$$

De la ecuación anterior, se quiere calcular la velocidad en el tiempo *n+1*. Para ello, primero se tiene que expresar la aceleración también de forma discreta. La aceleración de los nodos no es más que la fuerza exterior que actúa sobre cada uno de ellos dividida entre su masa. Por ello, la aceleración depende de los mismos factores que las que actúan sobre los nodos: de la posición y de la velocidad. Para mayor estabilidad, se resuelve la ecuación implícita mostrada en (2.113). Ésta es implícita porque se calcula la aceleración en el estado *n+1*, un estado que todavía no se ha determinado pues depende justamente de la velocidad que se desea calcular. Los vectores de posición, velocidad y aceleración son vectores columna de 3*m* dimensiones, donde m es el número de nodos del objeto.

$$\mathbf{v}^{\mathbf{n+1}}=\mathbf{v}^{\mathbf{n}}+\Delta t\mathbf{a}(\mathbf{x}^{\mathbf{n+1}},\mathbf{v}^{\mathbf{n+1}}) \tag{2.113}$$

Como se ha comentado, se quiere resolver las velocidades en *n*+1 de la ecuación anterior, pero no se tiene la aceleración en *n+1*. Se propone entonces realizar una aproximación de Taylor de primer orden, lineal, de la aceleración, ecuación (2.114). Esta aproximación se calcula tomando como referencia el estado *n*, el inicio del paso, por lo que se llega a (2.115).

$$\Delta\mathbf{a}(\mathbf{x},\mathbf{v})=\frac{\partial\mathbf{a}(\mathbf{x},\mathbf{v})}{\partial\mathbf{x}}\Delta\mathbf{x}+\frac{\partial\mathbf{a}(\mathbf{x},\mathbf{v})}{\partial\mathbf{v}}\Delta\mathbf{v} \tag{2.114}$$

$$\mathbf{a^{n+1}} = \mathbf{a^n} + \left.\frac{\partial \mathbf{a}(\mathbf{x},\mathbf{v})}{\partial \mathbf{x}}\right|_{n+\frac{1}{2}} \left(\mathbf{x^{n+1}} - \mathbf{x^n}\right) + \left.\frac{\partial \mathbf{a}(\mathbf{x},\mathbf{v})}{\partial \mathbf{v}}\right|_{n+\frac{1}{2}} \left(\mathbf{v^{n+1}} - \mathbf{v^n}\right) \qquad (2.115)$$

Sustituyendo la aceleración en *n+1*, de la ecuación anterior, en (2.113) y agrupando los términos que multiplican la velocidad en *n+1*, se obtiene la ecuación (2.116). Éste es un sistema de ecuaciones del que únicamente se desconocen las velocidades en *n+1*. Habrá tantas ecuaciones e incógnitas como nodos tenga la malla que representa el tejido, por lo que interesa resolver el sistema de forma eficiente. Al ser la matriz de coeficientes simétrica se aplica el método del Gradiente Conjugado (Baraff & Witkin, Large Steps in Cloth Simulation, 1998) ya implementado en una función de la librería *freecloth*.

$$\left(\mathbf{I} - \Delta t \left.\frac{\partial \mathbf{a}(\mathbf{x},\mathbf{v})}{\partial \mathbf{v}}\right|_{n+\frac{1}{2}}\right)\mathbf{v^{n+1}} = \mathbf{v^n} + \Delta t \left(\mathbf{a^n} + \left.\frac{\partial \mathbf{a}(\mathbf{x},\mathbf{v})}{\partial \mathbf{x}}\right|_{n+\frac{1}{2}} \left(\mathbf{x^{n+1}} - \mathbf{x^n}\right) - \left.\frac{\partial \mathbf{a}(\mathbf{x},\mathbf{v})}{\partial \mathbf{v}}\right|_{n+\frac{1}{2}} \mathbf{v^n}\right) \qquad (2.116)$$

Sin embargo, para calcular tanto los jacobianos como el término de la aceleración en el estado *n* se debe tener acceso al modelo de la dinámica interna del tejido, en este caso el modelo descrito en (Baraff & Witkin, Large Steps in Cloth Simulation, 1998). Como ya se ha comentado varias veces a lo largo del documento, este modelo lo implementa en *dpoSimulator* la librería *freecloth*. Afortunadamente, ésta permite el cálculo de los jacobianos de las fuerzas que actúan sobre los nodos y dichas fuerzas en el estado *n*. Multiplicando la ecuación (2.116) por la matriz de masas de los nodos, **M**, se obtiene la ecuación (2.117). Ésta es la ecuación que resuelve *dpoSimulator* mediante el método del Gradiente Conjugado para calcular las velocidades en el paso *n+1*. La matriz de masas es una matriz diagonal en la que el valor del elemento de la diagonal corresponde a la masa del nodo de esa fila. Esta matriz cumple las ecuaciones en (2.118).

$$\left(\mathbf{M} - \Delta t \left.\frac{\partial \mathbf{f}(\mathbf{x},\mathbf{v})}{\partial \mathbf{v}}\right|_{n+\frac{1}{2}}\right)\mathbf{v^{n+1}} = \mathbf{v^n} + \Delta t \left(\mathbf{f^n} + \left.\frac{\partial \mathbf{f}(\mathbf{x},\mathbf{v})}{\partial \mathbf{x}}\right|_{n+\frac{1}{2}} \left(\mathbf{x^{n+1}} - \mathbf{x^n}\right) - \left.\frac{\partial \mathbf{f}(\mathbf{x},\mathbf{v})}{\partial \mathbf{v}}\right|_{n+\frac{1}{2}} \mathbf{v^n}\right) \qquad (2.117)$$

$$\begin{cases} \mathbf{M_{ii}} = m_i \\ \mathbf{M_{ij}} = 0 \Leftarrow \mathrm{i} \neq \mathrm{j} \end{cases} \qquad (2.118)$$

El algoritmo de actualización de velocidades descrito en este apartado se implementa en la clase **WorldResponser**, documentado en el apartado 5.1.19.

Método alternativo de actualización de velocidades:

Previamente, se probó otro método de actualización de velocidades más sencillo y rápido, al ser explícito y evitar la resolución de un sistema de ecuaciones de grandes dimensiones. Éste se basaba en realizar la suposición de que la aceleración se mantenía constante durante todo el paso, con lo que se obtenía la ecuación (2.120) que permitía calcular las nuevas velocidades directamente a partir de las velocidades en el tiempo *n* y las velocidades medias. Sin embargo, resultó que, al aplicar este método, la simulación se volvía *inestable*.

$$\frac{\mathbf{v}^{n+1} - \mathbf{v}^{n}}{\Delta t} = \frac{\mathbf{v}^{n+1/2} - \mathbf{v}^{n}}{\frac{\Delta t}{2}} \tag{2.119}$$

$$\mathbf{v}^{n+1} = \mathbf{v}^{n} + 2(\mathbf{v}^{n+1/2} - \mathbf{v}^{n}) = 2\mathbf{v}^{n+1/2} - \mathbf{v}^{n} \tag{2.120}$$

En una simulación inestable, las velocidades de nodos cercanos son muy dispares, de direcciones parecidas pero sentidos opuestos. Esta perturbación provoca una onda que no se amortigua, debido al método de actualización de velocidades adoptado, llegando a un punto en el que no se encuentra solución para el siguiente paso utilizando el algoritmo de simulación de la dinámica interna, el descrito en (Baraff & Witkin, Large Steps in Cloth Simulation, 1998), y la simulación falla. El problema de este método de actualización de velocidades, al final descartado, es que se basa en una premisa errónea. No se puede suponer una aceleración constante en un paso en el que ocurre una colisión.

2.3 Descripción del simulador desarrollado

Este apartado se dedica a describir el programa desarrollado, *dpoSimulator*, que implementa el algoritmo descrito en profundidad en el apartado 2.1. El nombre de la aplicación es debido a su funcionalidad directa: simular objetos planos deformables, abreviado *dpo* en inglés.

Como el lector podrá apreciar, el desarrollo del simulador ha incluido tanto la implementación del algoritmo de detección y resolución de contactos y colisiones como el desarrollo de herramientas que permitan que el usuario del programa pueda montar sus propias escenas a simular. Por ello, primero se introduce la interfaz gráfica del programa y como moverse por él, en el siguiente apartado, y luego se explica de forma sistemática cómo puede el usuario crear sus escenas.

Sin embargo, se debe tener presente que la aplicación *no* es un fin en sí misma. El objetivo principal consiste en que el programa sirva de base para desarrollar un planificador de manipulación de objetos planos deformables, de tejidos. Por ello, en la última sección de este apartado, se da una guía a las clases y funciones principales del programa y se explica cómo se pueden utilizar como *cimientos* para desarrollar el planificador. Para una información más completa, se puede consultar directamente el código del programa, comentado en formato *Doxygen*.

2.3.1 Manual de usuario

Requisitos, compilación y ejecución del programa de simulación

Requisitos mínimos:

- El simulador *dpoSimulator* se ha desarrollado y ejecutado siempre en un sistema operativo Linux, en concreto en Ubuntu 8.10. No se ha estudiado la portabilidad del simulador a sistemas operativos diferentes, por limitación temporal y porque no formaba parte del alcance del proyecto. Sin embargo, se estima que tal tarea sería difícil, pues el simulador utiliza librerías mucho más utilizadas, y por lo tanto actualizadas, en Linux que en Windows.
- Se debe tener instalada la librería Qt3. Ésta se utiliza para crear la interfaz gráfica de interacción del usuario. Actualmente, se utiliza la versión 3.3 que viene incluida en el sistema operativo Ubuntu 8.10. En caso de que no esté incluida, se puede instalar desde una terminal con el comando:

```
sudo apt-get install libqt3-mt-dev
```

- Se debe tener instalada la librería Open Inventor. Ésta se utiliza para representar la escena de forma tridimensional. Se sabe que el programa funciona bien para la versión actual, la 2.1.5. Para instalar la librería se debe ejecutar el siguiente comando desde una ventana de terminal:

  ```
  sudo apt-get install libinventor0
  ```

- También se debe tener instalada la librería SoQt. Ésta permite utilizar simultáneamente y en un mismo programa interfaces de Open Inventor y Qt. La versión con la que funciona *dpoSimulator* actualmente es la 1.4.1. Para instalar esta librería, se debe introducir en una terminal el siguiente comando:

  ```
  sudo apt-get install libsoqt3-dev
  ```

- La última librería fundamental para la compilación y funcionamiento del programa es la *simage*. Ésta es utilizada por el programa para guardar imágenes del simulador. Actualmente se utiliza la versión 1.7.0. Para instalar esta librería, introduzca en una terminal la siguiente instrucción:

  ```
  sudo apt-get install libsimage-dev
  ```

Requisitos recomendables:

- Para poder compilar el programa, es posible que se necesiten compiladores y otras herramientas auxiliares que no vengan de serie con la distribución de Linux. Por ello, es recomendable instalar un paquete de herramientas esenciales para la compilación con la siguiente instrucción en una terminal:

  ```
  sudo apt-get install build-essential
  ```

- Para compilar la documentación del programa, y poder verla en formato html, se debe tener instalada el programa Doxygen, actualmente se utiliza la versión 1.5.8. Esto se puede hacer en la terminal con la instrucción:

  ```
  sudo apt-get install doxygen
  ```

  Una vez instalado Doxygen, se puede compilar la documentación ejecutando desde una terminal en el archivo raíz del programa la instrucción:

  ```
  doxygen Doxyfile
  ```

  Tras compilar la documentación, ésta se puede consultar abriendo con Mozilla Firefox el archivo *html/index.html*, dirección relativa al directorio raíz del programa.

- Como se explica en el apartado 2.3.2, el programa se ha desarrollado en el entorno de programación KDevelop, versión 3.5.3. Por ello, se recomienda que cualquier modificación se realice con el mismo programa, para ahorrarse los inconvenientes de migrar el proyecto a otro entorno. Para instalar el programa se debe ejecutar en una terminal la instrucción:

  ```
  sudo apt-get install kdevelop
  ```

  El proyecto se puede abrir desde KDevelop en *Project/Open Project…* y abriendo el archivo *pmdo.project* en el directorio raíz del programa.

Compilación del programa:

Para compilar el programa, simplemente se tiene que abrir una terminal e ir al directorio raíz del programa y ejecutar el comando:

```
make
```

Esto compilará el programa y lo enlazará, generando un archivo binario ejecutable *dpoSimulator*. Se puede eliminar todos los archivos binarios generados con el comando:

```
make clean
```

Ejecución del programa:

Tras compilar el programa, éste se puede ejecutar desde el directorio raíz del mismo en la terminal con la instrucción:

```
./dpoSimulator
```

Interfaz del simulador

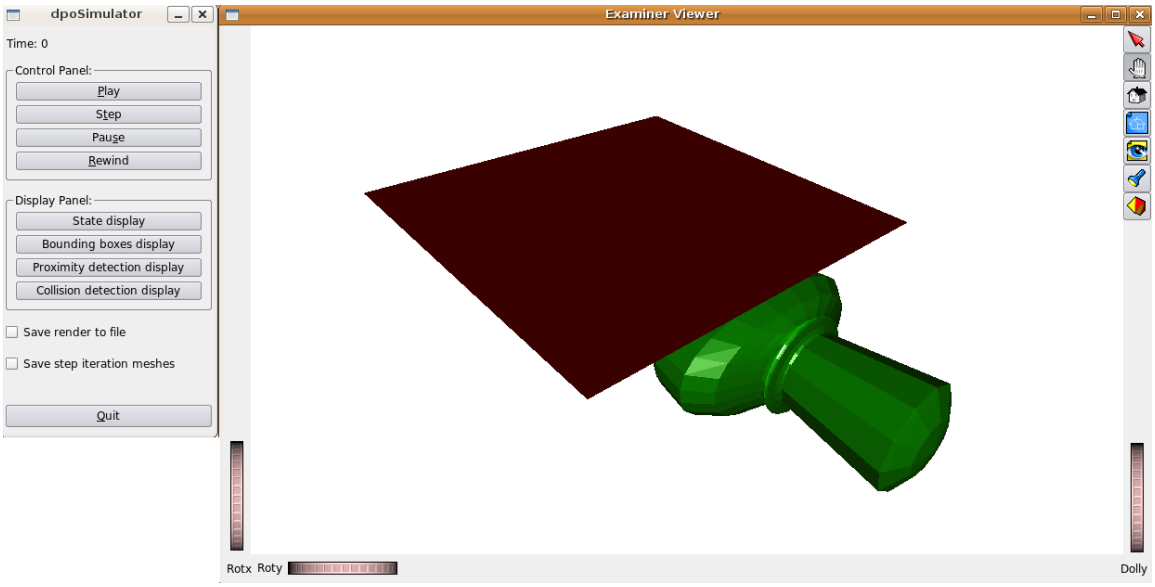


Figura 2.41: Al ejecutar el programa aparecen dos formularios, uno de control de la escena y el otro con la representación tridimensional de la misma.

La interfaz principal del programa consiste en dos ventanas:

- **Examiner Viewer:** Esta ventana, situada a la derecha en la Figura 2.41, permite ver la escena en un espacio tridimensional. Es un visualizador de Open Inventor que tiene ciertas funcionalidades adicionales, descritas más adelante.
- **Control Panel:** El panel de control es un formulario que permite al usuario controlar la evolución de la simulación y visualizar información adicional. Éste se muestra situado a la izquierda en la Figura 2.41. El formulario y la programación de eventos se ha desarrollado con Qt3. Más adelante, se explican las posibilidades y funcionamiento del panel de control.

La librería SoQt utilizada, permite la ejecución simultánea de un formulario de Qt3 con un visualizador de Open Inventor. A continuación, se describen estas dos ventanas y su funcionamiento.

Examiner Viewer

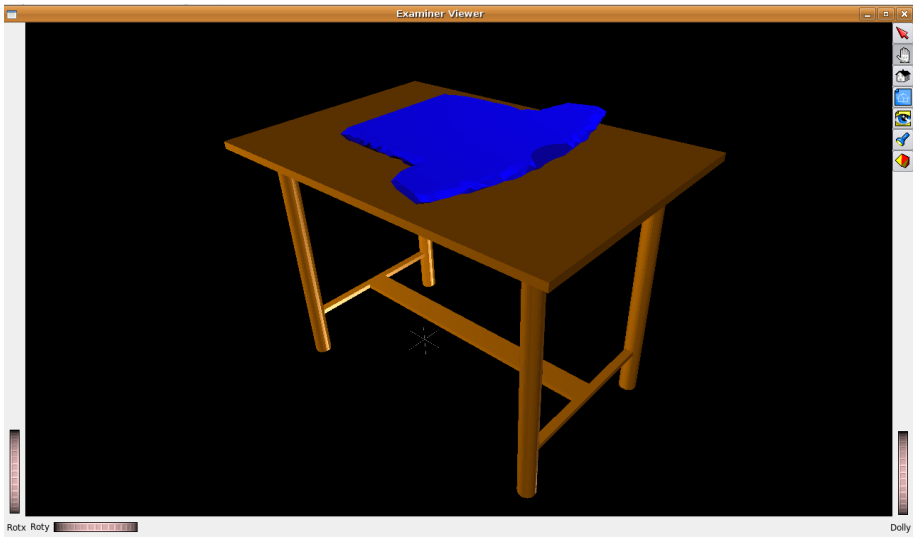


Figura 2.42: Ventana de visualización Examiner Viewer.

El *Examiner Viewer* es una ventana estándar de Open Inventor. Esta ventana permite al usuario del simulador:

- **Rotar la escena:** El cursor que aparece por defecto es el de rotación. Se puede rotar la escena simplemente manteniendo pulsado el botón izquierdo del mouse, en cualquier punto de la ventana y arrastrándolo.
- **Zoom:** Se puede hacer zoom rotando la ruedecita del mouse.
- **Desplazarse por la escena (*panning*):** Para desplazarse por la escena, se deben mantener pulsados el botón izquierdo y derecho del mouse mientras éste se mueve.
- **Centrar la vista:** Se puede centrar la vista en un punto concreto de una malla pulsando *s*, lo que cambiará el cursor del mouse a una cruz, y seleccionando el punto sobre el que se quiere centrar la vista. Centrar la vista en un punto permite que todas las rotaciones de la escena se hagan entorno a ese punto. Además, se podrá aumentar más, zoom, la escena entorno al punto seleccionado.
- **Cambiar el estilo de visualización:** La ventana *Examiner Viewer* permite visualizar la geometría de formas diferentes. Para ello, se debe pulsar el botón derecho del ratón en un punto cualquiera de la escena. Aparecerá un menú contextual en el que se pueden seleccionar diferentes estilos de visualización y animación, entre otras opciones. Cabe destacar las tres opciones de visualización siguientes, que se encuentran en *DrawStyles/StillDrawStyle*:
    - **Estilo de visualización estática *As Is*:** Este estilo es el seleccionado por defecto. Éste, mostrado en la Figura 2.43, muestra los triángulos de las mallas de forma sólida.

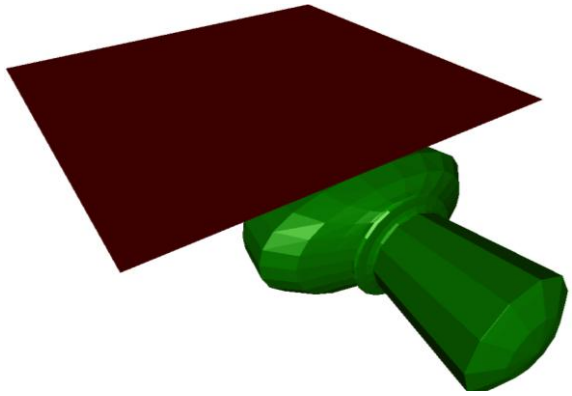


Figura 2.43: Escena *ClothOnBottle* mostrada con el estilo de visualización *As Is*.

- **Estilo de visualización *Wireframe*:** Este estilo sólo muestra las aristas coloreadas. De esta forma, los objetos no ocultan que hay detrás. De esta forma, este estilo resulta muy útil para visualizar contactos y colisiones.

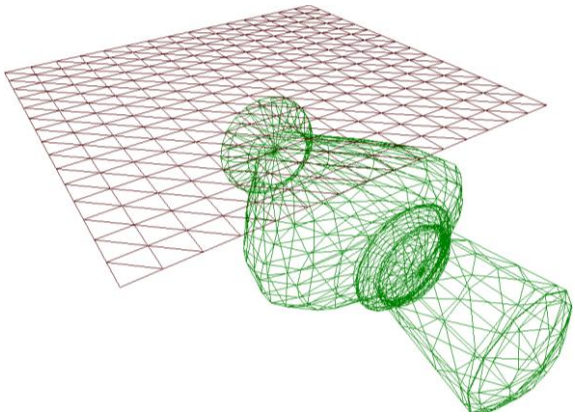


Figura 2.44: Escena *ClothOnBottle* mostrada con el estilo de visualización *Wireframe*.

- **Estilo de visualización *Wireframe Overlay*:** Este estilo es un híbrido de los dos anteriores. Es igual que el modo *As Is*, pero con las aristas resaltadas en color rojo.

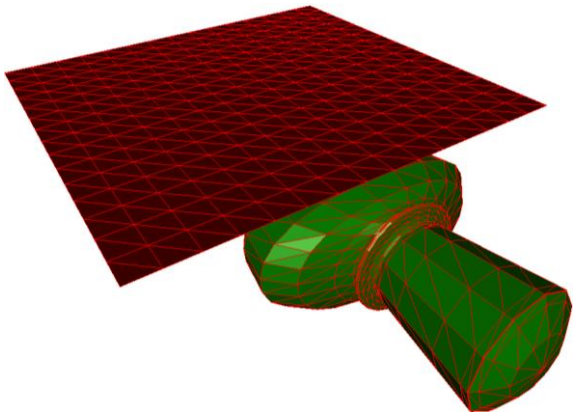


Figura 2.45: Escena *ClothOnBottle* mostrada con el estilo de visualización *Wireframe Overlay*.

Control Panel

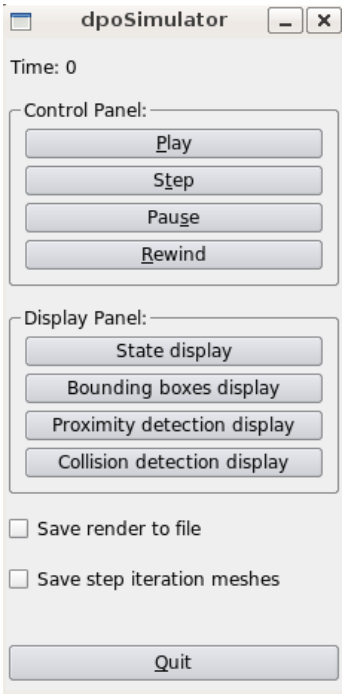


Figura 2.46: Panel de control del simulador.

El panel de control permite controlar el flujo de la simulación y mostrar información auxiliar. Arriba de todo se muestra el tiempo de la simulación. Éste siempre empieza en *0*, como se muestra en la Figura 2.46. Las opciones básicas de control de la simulación son:

- **Play:** Esta instrucción ejecutará el simulador de forma cíclica, avanzado la simulación automáticamente. Se observará como el tiempo va progresando y la escena va cambiando.
- **Step:** Pulsar este botón avanza la simulación en un solo paso.
- **Pause:** Permite interrumpir la simulación mientras se estaba ejecutando, habiendo pulsado *play*. Si una vez pausado se vuelve a pulsar *play*, la simulación continuará desde el punto en el que se interrumpió.
- **Rewind:** Esta instrucción reinicia la escena y la deja en pausa.
- **Quit:** Permite cerrar el simulador.

A continuación se describen las opciones avanzadas del simulador, que permiten comprobar el correcto funcionamiento del mismo y obtener información adicional:

a) State Display:

Figura 2.47: Panel de visualización del estado de la escena.

Pulsar en el panel de control el botón *State display*, abre la ventana de opciones de visualización del estado que se muestra en la Figura 2.47. En este se puede escoger las siguientes opciones:

i) Display Vertices:

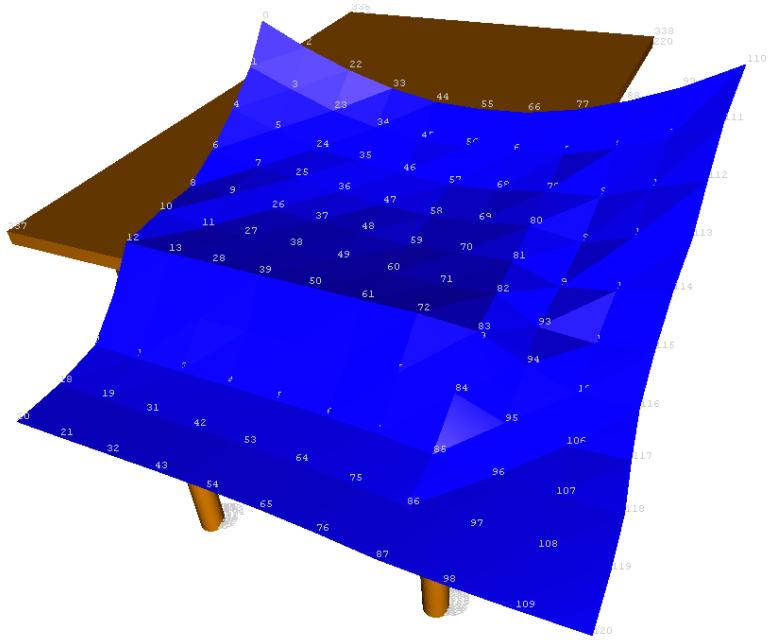


Figura 2.48: Escena con vértices etiquetados.

La visualización de vértices permite ver etiquetados los vértices con el número de identificación que tienen en su malla. Esto es útil para manipular manualmente las velocidades de los vértices pues se ve a simple vista el identificador del vértice que se quiere manipular.

ii) Display Speed:

Esta opción permite mostrar las velocidades de los nodos en el estado actual de la simulación. Éstas se representan como flechas que parten de los nodos cuyas velocidades representan, ver Figura 2.49.

Hay dos parámetros adicionales, mostrados en la Figura 2.47, que se habilitan al mostrar las velocidades y se pueden manipular para modificar la representación de las flechas que representan los vectores de velocidad:

- **Length factor:** Variar este parámetro varía la longitud de las flechas, de forma proporcional. Si se escogiese un tamaño fijo para las flechas, según las dimensiones de la escena el usuario no podría verlas bien, pues serían demasiado cortas o largas. En ese caso, el usuario puede aumentar o disminuir el parámetro *length factor* hasta que se vean bien las flechas.
- **Width factor:** Este parámetro es el análogo al anterior aplicado al ancho de las flechas. Cuanto mayor es el factor, mayor es el radio de las flechas.

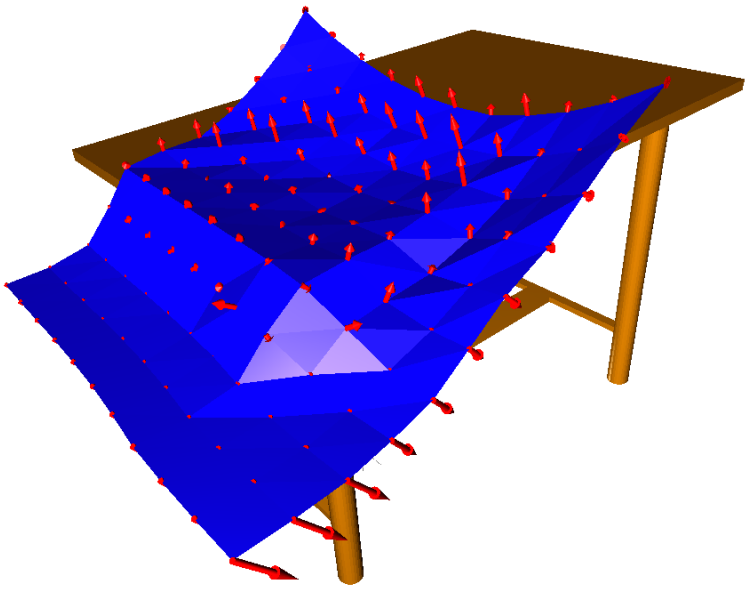


Figura 2.49: Velocidades mostradas como flechas rojas en la escena *ClothOnTable*.

La representación de velocidades ha resultado ser esencial para el desarrollo de la simulación, pues permitió descartar un método de actualización de velocidades considerando la aceleración constante al ver que se producía una inestabilidad en las velocidades, para más información consulte la página 47.

La visualización del campo de velocidades permite juzgar el comportamiento del simulador. Para la escena representada en la Figura 2.49 se pueden realizar las siguientes observaciones:

- Las velocidades de los nodos con movimiento restringido, las dos esquinas superiores de las que está colgando el tejido azul, son nulas. Esto es correcto, pues tal es la restricción de velocidad que se ha aplicado, se puede comprobar en el directorio *environment_ClothOnTable/bodies/vel_ClothOnTable*.
- Se observa que los nodos inferiores del tejido tienen una velocidad descendente, debido a la caída que van a sufrir.
- Los nodos del tejido que se encuentra sobre la mesa tienen una velocidad ascendente. Esto se debe a que ya han entrado en contacto con la mesa y ahora van a subir debido a la tracción.

b) Bounding Box Display:

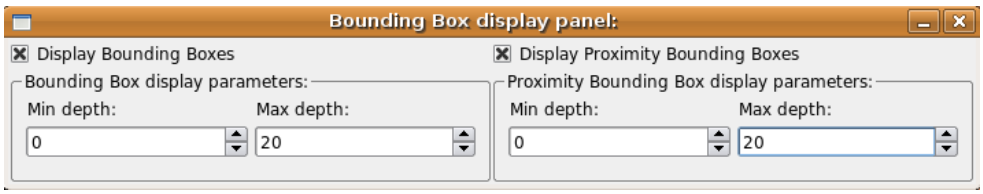


Figura 2.50: Panel de visualización de cajas delimitadoras.

Este panel permite ver los árboles de cajas delimitadoras, Bounding Box Trees, normales y de contacto en todos sus niveles. Para más información sobre cajas delimitadoras y su función en el simulador consulte la sección 2.2.3.

i) ***Display Bounding Boxes:***

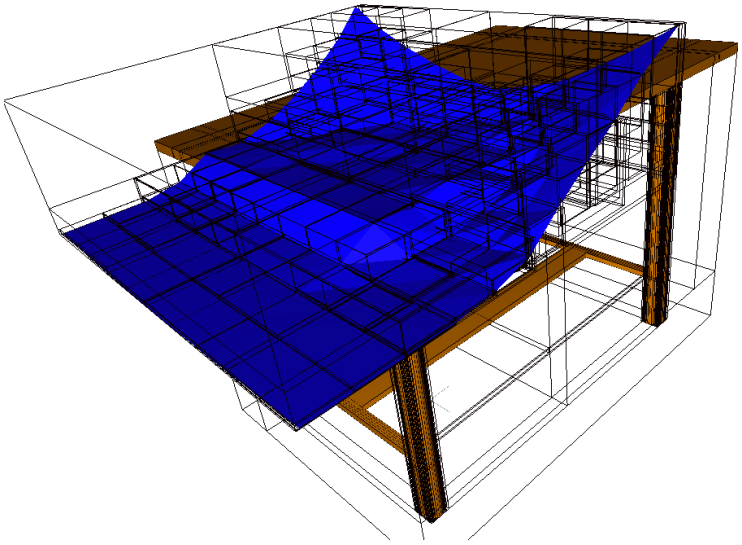


Figura 2.51: Escena con todos los niveles de los árboles de cajas delimitadoras mostrados en *dpoSimulator*.

Las cajas delimitadoras se representan como cubos alámbricos, ver la Figura 2.51, negros. El panel permite determinar el rango de niveles, dentro de los árboles de cajas delimitadoras, que se muestran. La Figura 2.51 muestra todos los niveles, por lo que se muestran todas las cajas delimitadoras. Las dos cajas más grandes, una delimita todo el tejido azul y la otra delimita toda la mesa, se solapan. Por ello, una primera detección aproximada, ver el apartado 2.2.3ya determina que puede haber colisiones y contactos en la escena. La Figura 2.52 muestra todos los ocho primeros niveles de la escena. Para que sólo se muestre el quinto nivel, por ejemplo, se pone en el panel que se muestren los niveles desde 5 hasta 5.

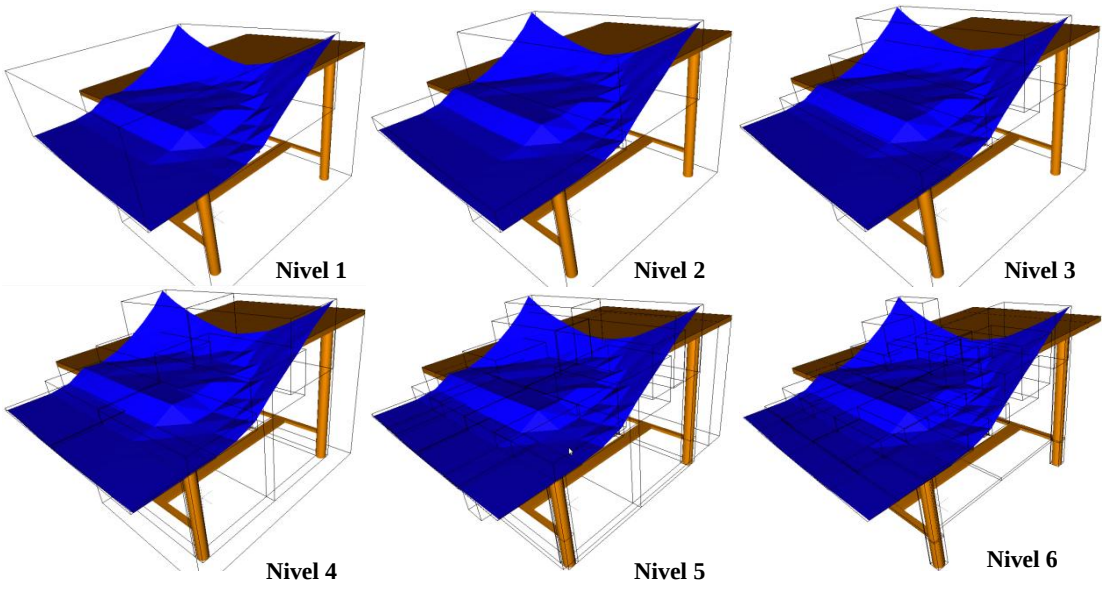


Figura 2.52: De izquierda a derecha y arriba a abajo, se muestran los primeros seis niveles de los árboles de cajas delimitadoras de la escena ClothOnTable. Las imágenes han sido tomadas del simulador *dpoSimulator*.

ii) Display Proximity Bounding Boxes:

Siguiendo el mismo procedimiento que en el caso anterior, se pueden visualizar las cajas delimitadoras utilizadas para la detección de contactos. Éstas no solo delimitan la geometría, sino que consideran que ésta tiene un pequeño espesor, por lo que son más grandes. Para más información sobre las cajas delimitadoras para contactos, consulte el apartado 2.2.3.

La Figura 2.53, muestra todos los niveles de las cajas delimitadoras para contactos de la escena *ShirtOnTable*. La camiseta tiene una malla muy tupida, por lo que hay muchas cajas delimitadoras. Sin embargo, se observa que la superficie de la mesa es menos tupida, pues sólo necesita dos triángulos para ser representada, acelerando el cálculo. Las patas son muy tupidas, puesto que son cilindros representados con triángulos. Sin embargo, no ralentizan mucho el cálculo, pues una comprobación rápida, con las cuatro cajas que delimitan las cuatro patas, las rechaza como posibles objetos que estén en contacto con la camisa. Se hace notar al lector, que la visualización de las cajas delimitadoras de contacto es muy parecida a las cajas delimitadoras normales, pues la camiseta es poco gruesa, 3mm. Sin embargo, es importante tener en cuenta el sobrespesor para que el simulador trate todos los contactos.

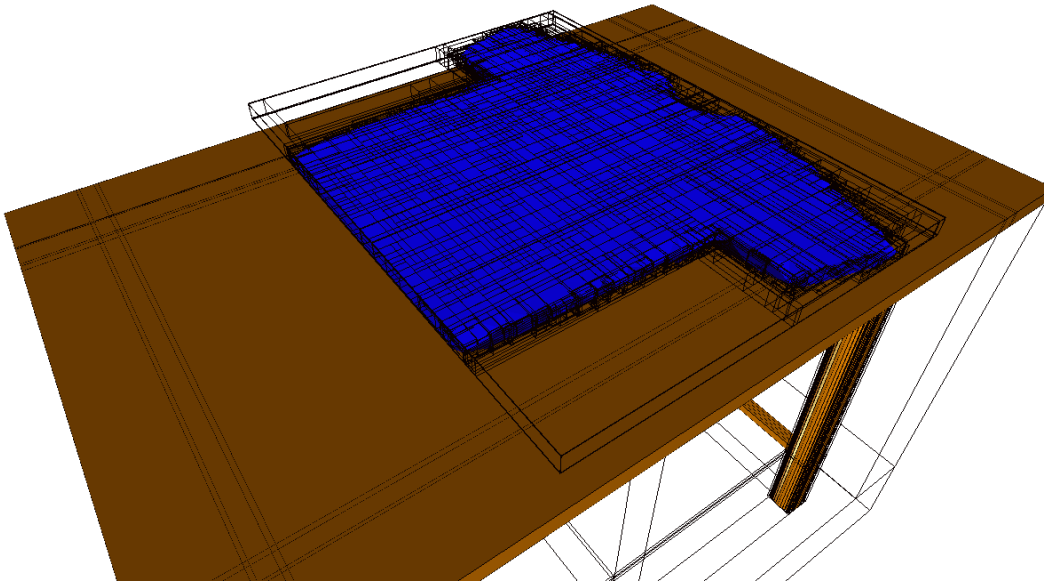


Figura 2.53: Árboles de cajas delimitadoras completos para la detección de contactos en una escena.

c) Proximity Detection Display:

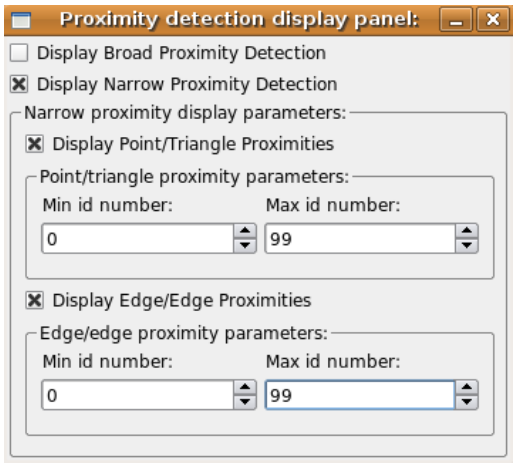


Figura 2.54: Panel de visualización de contactos.

El panel de visualización de contactos permite que el usuario pueda comprobar el funcionamiento del algoritmo de detección de contactos. Como se explica en el apartado 2.2.3, la detección de contactos tiene dos fases, una aproximada y una precisa. Este panel permite ver el resultado de las dos fases de detección de contactos:

i) ***Display Broad Proximity Detection:***

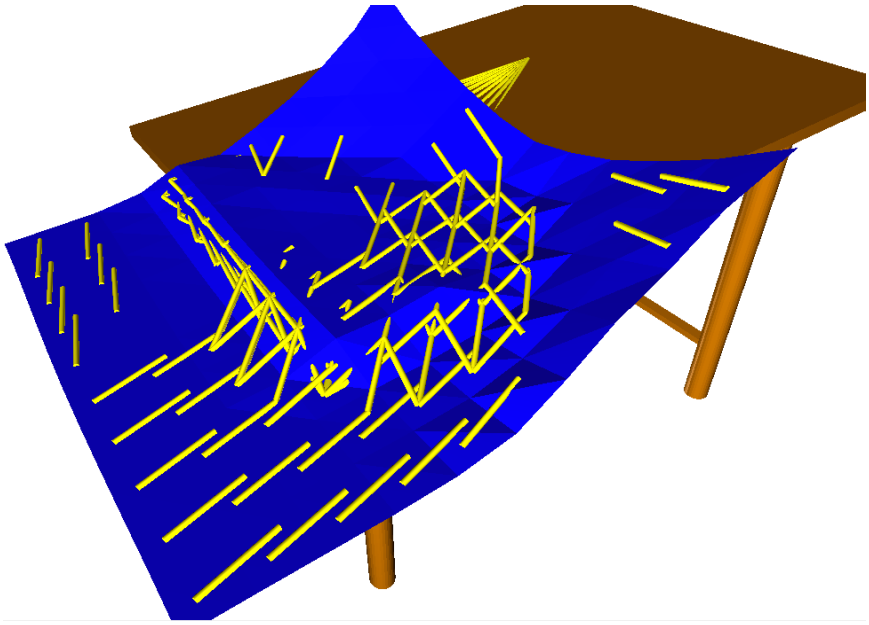


Figura 2.55: Resultado de una detección aproximada de contactos aplicada a la escena *ClothOnTable*.

La detección aproximada de contactos se basa en la comprobación inteligente de solapamientos de cajas delimitadoras. Esta detección genera una lista de parejas de entidades que *es posible* que estén en contacto. Las parejas de *posibles* entidades en contacto se muestran mediante una línea amarilla que une los centros de las dos entidades, como se puede muestra en la Figura 2.55 y la Figura 2.56. Se destaca que el número de contactos detectados de forma aproximada reduce mucho la lista de posibles entidades en contacto.

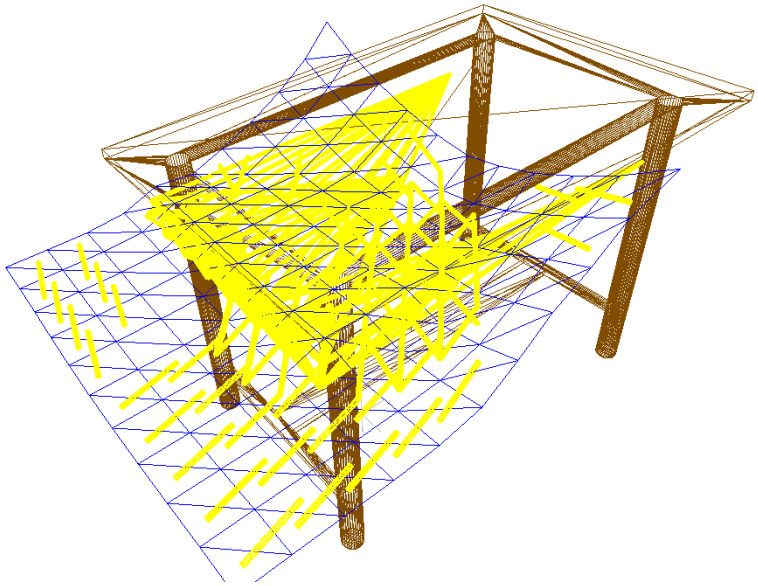


Figura 2.56: La representación alámbrica de la geometría permite observar los contactos aproximados entre la tela y la mesa.

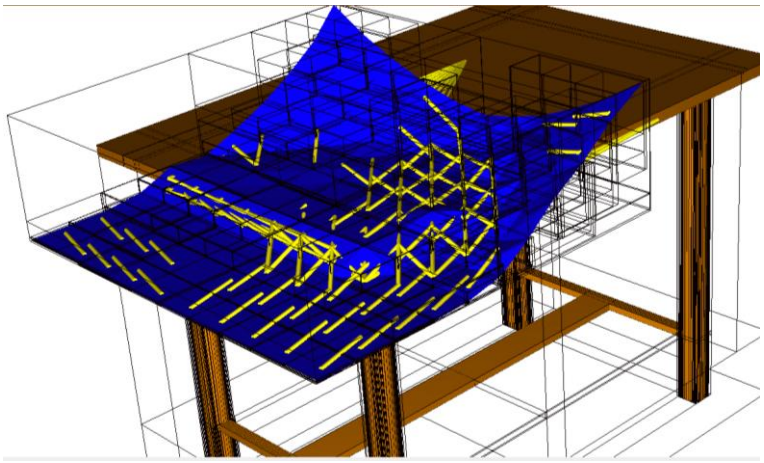


Figura 2.57: La detección aproximada de contactos se basa únicamente en las cajas delimitadoras de contacto.

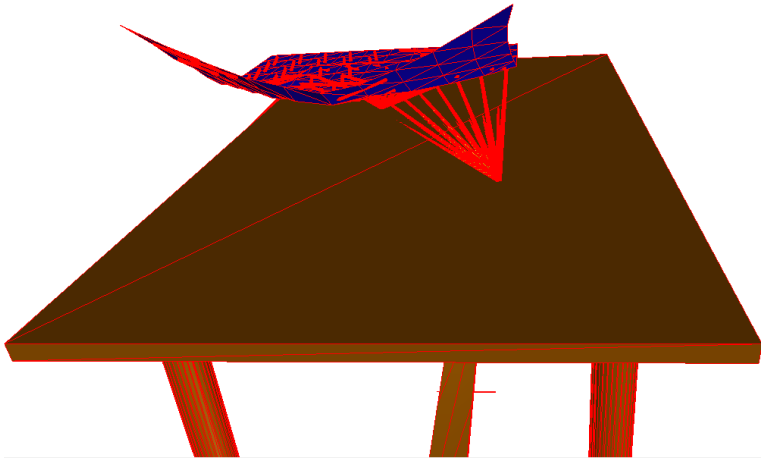


Figura 2.58: La superficie de la mesa se representa con dos triángulos.

Realizando una simple inspección visual, se observa que, en la escena *ClothOnTable*, hay tres tipos de pareja preseleccionadas para aplicar contactos:

1. Entidades de la ropa que es posible que estén en contacto con los dos triángulos que componen la superficie de la mesa. Estas parejas se pueden ver bien en la Figura 2.58. Van desde algunos triángulos de la malla del tejido al centro alguno de los dos triángulos de la superficie de la mesa.
2. Entidades de la ropa que pueden estar en contacto con el borde de la mesa. Estas parejas se pueden ver claramente en la Figura 2.56.
3. Entidades del tejido que pueden estar en contacto con otras entidades del tejido. Éstas se muestran como líneas amarillas que van desde el centro de algunos triángulos de la malla del tejido a otros triángulos de la misma malla. Una simple inspección visual determina que estos triángulos no están en contacto. Sin embargo, es necesario detectar contactos de forma aproximada entre triángulos de una misma malla, pues el tejido puede entrar en contacto consigo mismo. Una detección precisa ya eliminará todos estos falsos positivos.

Todas las parejas de triángulos, que posiblemente estén en contacto, mostrados en la Figura 2.59, se corresponden al tercer caso. Todos estos falsos positivos se dan porque inicialmente la malla del tejido es perfectamente horizontal y las cajas delimitadoras de los triángulos se tocan en las esquinas. Todas estas parejas de posibles contactos se descartarán en la detección precisa.

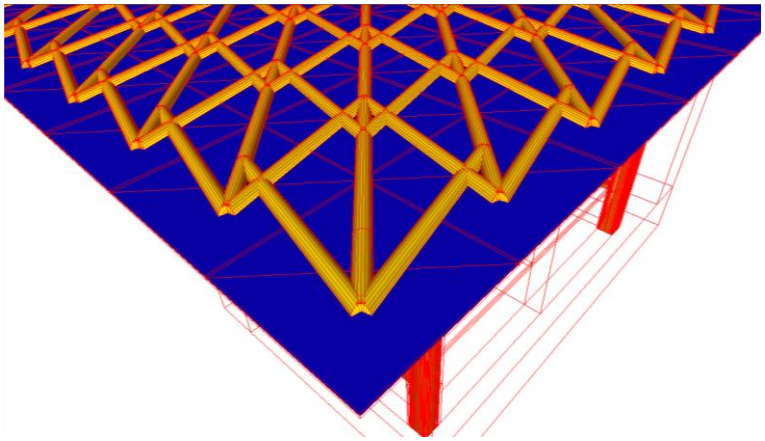


Figura 2.59: Inicialmente, se detectan, de forma aproximada, muchos contactos debido a que las cajas delimitadoras de contacto de la ropa se tocan en las esquinas.

ii) Display Narrow Proximity Detection:

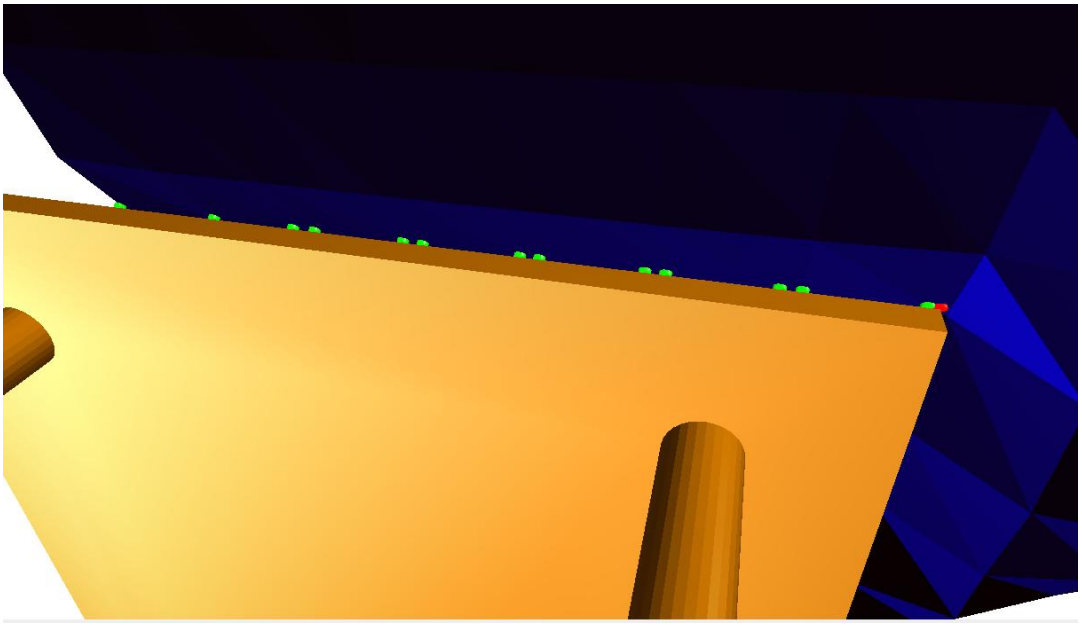


Figura 2.60: Se muestran en verde los contactos entre aristas y en rojo los contactos nodo-triángulo.

La Figura 2.60 muestra los contactos detectados, de forma precisa, entre el tejido y la mesa para la misma escena, y el mismo momento, que la mostrada en la Figura 2.55. Estos 14 contactos son los únicos que no se han descartado de todos los detectados de forma aproximada en la Figura 2.55. Como se explica en el apartado 2.2.3, hay dos tipos de contacto:

- **Contactos entre aristas:** Estos se muestran como una línea de color verde que une los dos puntos de las aristas más cercanos, y que están a una distancia menor al grosor del tejido. En la escena representada en la Figura 2.60 hay 13 contactos entre la arista que representa el borde de la mesa y aristas del tejido.
- **Contactos entre un nodo y un triángulo:** Estos se representan con una línea de color rojo que une el nodo y el punto del triángulo más cercano al nodo. En la Figura 2.60 sólo hay un contacto de este tipo, entre la esquina de la mesa y un triángulo del tejido.

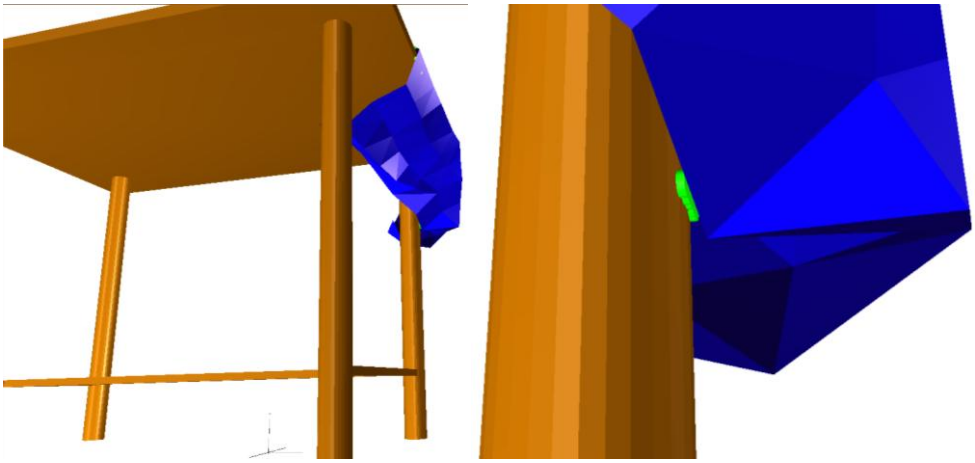


Figura 2.61: Contactos de tipo arista-arista entre el tejido y una pata de la mesa.

Se puede seleccionar los contactos que se quiere mostrar, seleccionando el rango de identificadores de los contactos a mostrar en el panel de visualización de contactos, Figura 2.54. Esto ha sido útil en el desarrollo del simulador para comprobar que los contactos se habían detectado de forma correcta.

d) Collision Detection Display:

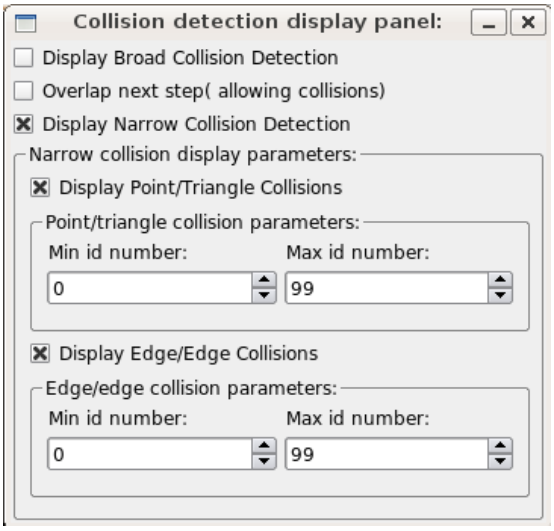


Figura 2.62: Panel de visualización de colisiones detectadas.

Igual que para la detección de contactos, hay un panel que permite visualizar las colisiones detectadas en un paso. Éste tiene las siguientes opciones:

i) Overlap next step (allowing collisions):

Esta opción, no disponible en el panel de detección de contactos, permite ver cuál sería el estado siguiente de simulación si no se aplicasen impulsos debidos a la colisión. Por ello, se puede ver en la Figura 2.63 que los objetos se atraviesas sin que haya ninguna interacción entre ellos. También se observa que el estado siguiente se muestra con un color más saturado que el actual. Se hace notar que en el caso la mesa, al estar quieta, el estado siguiente es igual al actual, por eso aparece con un marrón más claro.

La representación de la estimación del siguiente estado es útil para comprobar si la detección de colisiones se realiza de forma correcta.

Figura 2.63: Escenas *ClothOnTable*, izquierda, y *HangingCloth2*, derecha, en las que se ha superpuesto el siguiente estado estimado.

ii) Display Broad Collision Detection:

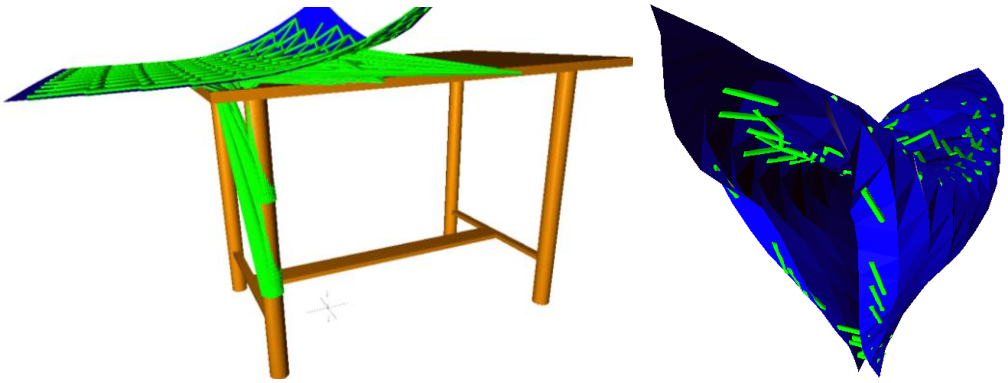


Figura 2.64: Detección aproximada de colisiones de la escena *ClothOnTable*, izquierda, y *HangingCloth2*, derecha.

La representación de la detección aproximada de colisiones se realiza de la misma forma que su homóloga para contactos. Unas líneas, cilindros finos, verdes unen los centros de los triángulos que es posible que colisionen durante el paso que se va a simular. En la imagen derecha de la Figura 2.64, se ve claramente que las colisiones que van a ocurrir, entre los triángulos de las esquinas que inferiores del tejido, están incluidas en la lista de *posibles* colisiones.

iii) Diplay Narrow Collision Detection

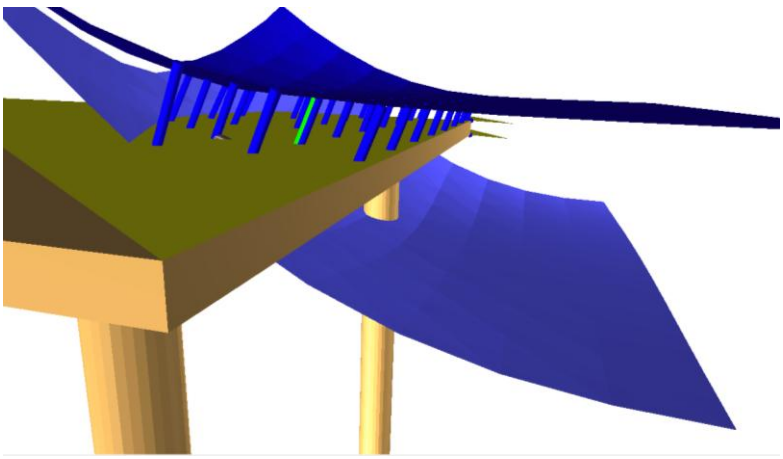


Figura 2.65: Detección precisa de colisiones en *ClothOnTable* con la estimación del siguiente estado superpuesta.

También se pueden representar gráficamente las colisiones que se han detectado de forma precisa. La Figura 2.65 muestra las colisiones detectadas entre un estado y el siguiente, solapado. Hay dos tipos de colisiones:

- **Entre un nodo y un triángulo:** Se representa, con un cilindro azul, la trayectoria del nodo, supuesta lineal, desde el principio del paso hasta que ocurre la colisión. También se representa la posición del triángulo cuando ocurre la colisión, de color amarillo. La imagen izquierda de la Figura 2.66 muestra las colisiones que ocurren, en el mismo caso que el mostrado en la Figura 2.65, entre el nodo de la esquina de la mesa y un triángulo de la malla del tejido. La imagen derecha de la misma figura es más clara, pues sólo se muestra una colisión entre el nodo de una esquina con el triángulo de otra del mismo tejido.

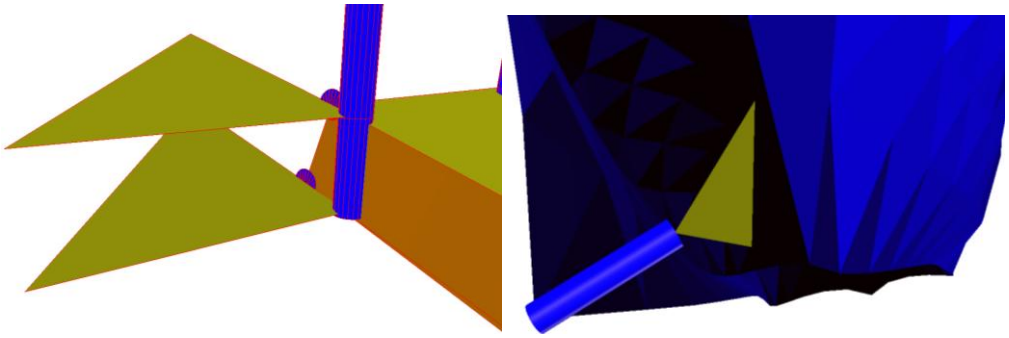


Figura 2.66: Colisión nodo triángulo para la escena *ClothOnTable*, izquierda, y *HangingCloth2*, derecha.

- **Entre dos aristas:** Dos cilindros grises representan la posición de las dos aristas en el momento de la colisión y un cilindro verde une los puntos de las aristas que van a colisionar al principio del paso, ver Figura 2.67.

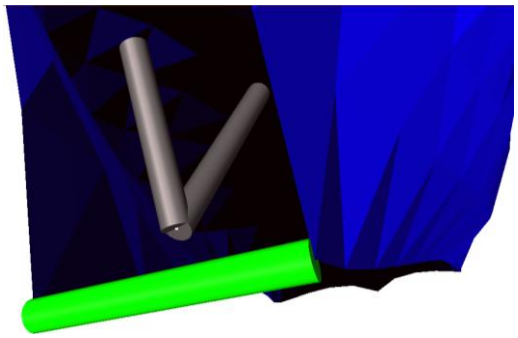


Figura 2.67: Colisión entre dos aristas de una misma malla.

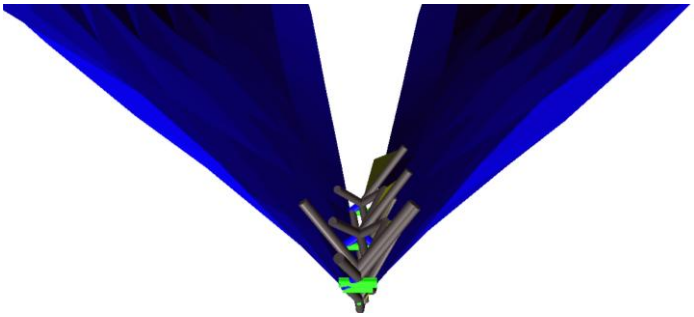


Figura 2.68: Representación de todas las colisiones que ocurren en un paso de la escena *HangingCloth2*.

e) Save render to file:

Una vez activada esta opción, todos los pasos que se simulen se guardarán, como imágenes, en la carpeta *frames* del directorio raíz del programa. Al activar la opción, el programa borrará la carpeta, en caso de que ya existiese y la volverá a crear. Las imágenes se guardarán de forma consecutiva con el nombre *frame000000.jpg*, *frame000001.jpg*, *frame000002.jpg*, etc. La Figura 2.69 muestra las imágenes guardadas en intervalos de 5 ms de la simulación de la escena *HangingCloth2*.

Guardar imágenes consecutivas con esta opción resulta muy útil para grabar videos. El siguiente comando, en una terminal de Linux, ejecutado desde la carpeta *frames*, genera un video *movie.avi*, de 10 fps, con todas las imágenes guardadas en dicho directorio:

```
mencoder "mf://*.jpg" -mf fps=10 -o movie.avi -ovc lavc -lavcopts
vcodec=msmpeg4v2:vbitrate=800
```

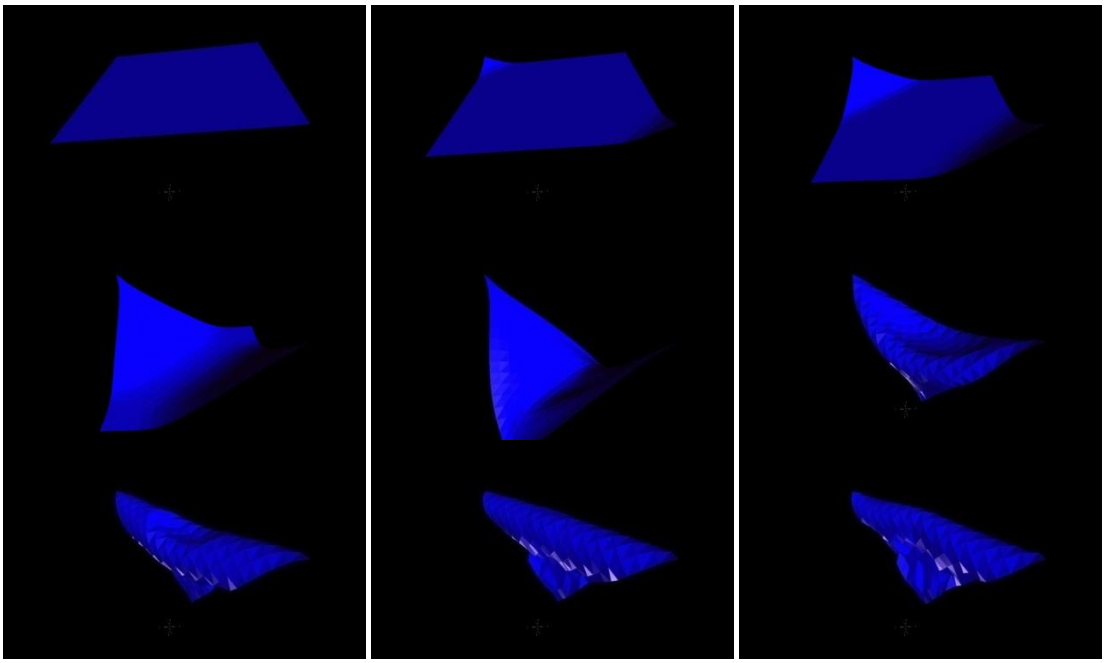


Figura 2.69: De la izquierda a derecha y arriba a abajo, se muestra la evolución de caída de un trozo de ropa colgado de dos esquinas en intervalos de 5 milisegundos, se corresponde a la escena ClothHanging2. Las imágenes se han guardado verificando la casilla "Save render to file"

f) Save step iteration meshes:

De la misma forma que el simulador permite guardar imágenes, *dpoSimulator* permite guardar *todas* las estimaciones del estado siguiente que se generan en las iteraciones que realiza el algoritmo de resolución de colisiones, para más información consulte el apartado 2.2.1. Las mallas quedan guardadas en la carpeta *iterationMeshes*, en el directorio raíz del programa, en formato stl.

Se ha desarrollado un programa auxiliar llamado *stlViewer* que permite ver todo un grupo de mallas, con colores aleatorios, pero que no realiza ninguna simulación. *stlViewer* es un simple visualizador de mallas en formato stl. Resulta muy útil representar todas las mallas guardadas por *dpoSimulator* en un paso determinado con *stlViewer*. Para ello, se debe ejecutar la siguiente instrucción desde una terminal en el directorio raíz:

```
./stlViewer iterationMeshes/mesh*
```

La Figura 2.70 muestra las mallas guardadas en un paso de la simulación de la escena *HangingCloth2*. Hay dos mallas casi perfectamente superpuestas, una ocre y la otra verde. Son

iguales en todas partes excepto en la zona inferior, donde ocurre la colisión. En esta zona inferior se observa que predomina el color verde, pues la malla verde oculta la ocre en esta zona. La malla verde se corresponde a una iteración anterior a la ocre. Por eso la malla verde presenta una colisión y la ocre no, pues es la iteración final en la que la colisión ya se ha resuelto.

Figura 2.70: *stlViewer* permite ver todas las mallas guardadas por *dpoSimulator*.

Construcción de una escena

El simulador busca la descripción de la escena a simular en la carpeta *environment* que se encuentra en el directorio raíz del programa. En el CD adjunto a este documento, se hallan varias carpetas con escenas diferentes. Cambiando el nombre de una de estas carpetas a *environment*, el programa cargará la escena indicada.

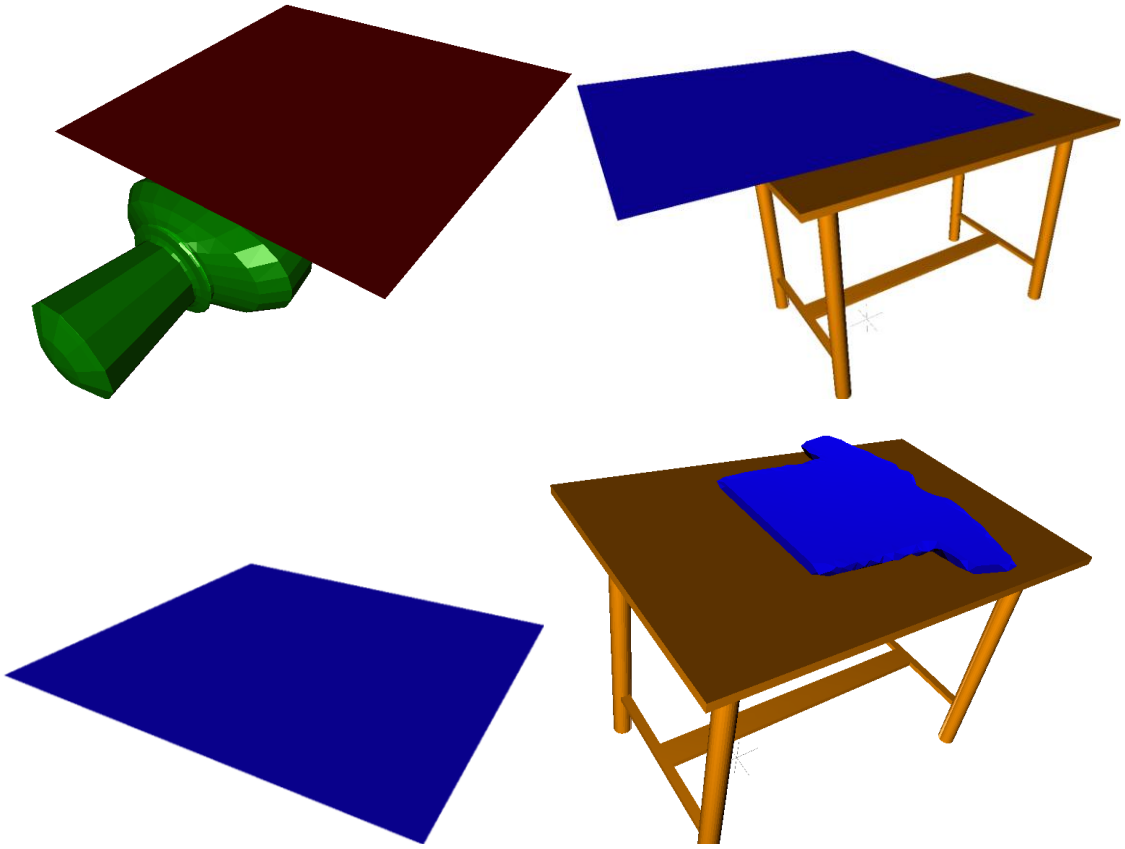


Figura 2.71: De arriba a abajo e izquierda a derecha se muestran los estados iniciales de las escenas *ClothOnBottle*, *ClothOnTable*, *HangingCloth2* y *ShirtOnTable*

Evidentemente, el usuario puede crear sus propias escenas. En este apartado, y los que siguen, se explica cómo hacerlo.

Configuración de los parámetros de la escena

En el directorio *environment/world_parámeters* se encuentra el fichero *world_parameters*. Abriendo este con un editor de texto, se muestra un fichero de configuración de los parámetros de la escena parecido al siguiente:

```
World parameters:

Please only change the values of the world parameters. Do not
change the
order and do not delete a parameter.

Gravity (m/s^2) = 9.81

Framerate (frames/second) = 100

End of file.
```

En este documento, el usuario puede cambiar la gravedad y el número de estados intermedios que se calculan y se muestran para cada segundo real, en este caso el simulador calcularía y mostraría estados en intervalos de 0,01 segundos.

Lista de objetos de la escena

Los objetos y sus propiedades se pueden añadir a la escena en el documento *body_list* del directorio *environment/bodies* parecido al que se muestra a continuación:

```
Here the user can add the files to be taken into account when
loading bodies. The files can start with: dpo (if they contain
information about a deformable planar object), rig (if they
contain information about a rigid object) or vel ( if they
contain information about velocity constraints which affect
objects). NOTE: vel files must be located after the files
that contain the dpo or rig objects that they refer to.

Start the list from here:
dpo_ClothOnTable
par_ClothOnTable
vel_ClothOnTable
rig_table
par_table
vel_table
```

En este fichero *body_list* hay dos objetos, el objeto *ClothOnTable y el table*, cada uno de los cuales se describe con tres documentos, que contienen, en este orden:

- **Descripción de la malla del objeto:** Este documento contiene la malla del objeto en formato STL. Más adelante se indica cómo se puede importar mallas desde CATIA. Si el objeto es deformable plano, un tejido, el nombre del fichero que contiene la descripción de su malla debe empezar por *dpo*, del inglés ***d****eformable* ***p****lanar* ***o****bject*, y si es un objeto rígido debe empezar por *rig*. Se debe indicar de esta forma si el objeto es deformable o rígido pues la descripción de su malla es igual, en formato STL. En el caso mostrado más arriba, el objeto *ClothOnTable* es de tipo dpo, *dpo_ClothOnTable*, y el objeto *table* es de tipo rígido, *rig_table*.
- **Parámetros del objeto:** Los parámetros del objeto se configuran en un documento que tiene el nombre del objeto prefijado por *par*. Éste es igual para objetos deformables y rígidos. Más adelante se explica cómo cambiar los parámetros de los objetos. En el caso anterior, hay un archivo de parámetros para cada objeto, *par_ClothOnTable* y *par_table*.

- **Velocidades restringidas del objeto:** Las restricciones de velocidades de algunos nodos de un objeto se encuentran en un documento con el nombre del objeto prefijado con *vel*. Más adelante se explica cómo se encuentra la información en este documento. En el caso anterior, cada objeto tiene su documento de velocidades *vel_ClothOnTable* y *vel_table*.

Los documentos a los que se hace referencia en *body_list* deben estar en la carpeta *environment/bodies*. Se hace notar al lector que no todos los objetos descritos en esta carpeta se mostrarán en la simulación, pues sólo lo harán los que además estén incluidos en la lista de objetos, *body_list*.

Definición de las mallas

El formato STL puede estar en ASCII o binario. El simulador sólo acepta documentos STL en formato ASCII, que son los que genera CATIA por defecto. Éstos contienen una lista de triángulos. Para cada triángulo, muestran primero el vector normal y luego la posición de cada vértice. El simulador identifica todos los vértices que son iguales y los considera como uno solo, pues es necesario que no haya vértices repetidos para que la simulación se ejecute correctamente. A continuación se muestra el documento que describe la malla *simpleCloth*.

```
solid simpleCloth
  facet normal  0.000000e+00  -0.000000e+00  1.000000e+00
    outer loop
      vertex    -5.000000e-01  -5.000000e-01  1.200000e+00
      vertex    5.000000e-01  -5.000000e-01  1.200000e+00
      vertex    -5.000000e-01  5.000000e-01  1.200000e+00
    endloop
  endfacet
  facet normal  -0.000000e+00  0.000000e+00  1.000000e+00
    outer loop
      vertex    -5.000000e-01  5.000000e-01  1.200000e+00
      vertex    5.000000e-01  -5.000000e-01  1.200000e+00
      vertex    5.000000e-01  5.000000e-01  1.200000e+00
    endloop
  endfacet
endsolid simpleCloth
```

Esta malla es sencilla y se podría haber hecho a mano, pero la generación de documentos STL de mallas más complejas, como las mostradas en la Figura 2.72, requerirán métodos más automatizados. Más adelante, en este mismo apartado, se explica cómo importar mallas desde CATIA en formato STL. Sin embargo, se pueden generar mallas planas y rectangulares, para objetos deformables planos, con un programa auxiliar desarrollado llamado *fabricCreator*. Si este ejecuta, sin ningún parámetro, desde una ventana de terminal en el directorio raíz del programa, se muestra el siguiente mensaje en el que se indica qué hace el programa y qué parámetros se le debe pasar:

```
This program creates a rectangular planar mesh and saves it in a
file in stl format.

Command: ./fabricCreator <filename> <meshName> <width> <length>
<wDivisions> <lDivisions> <xcm> <ycm> <zcm>
(Length units in m.)
```

Los parámetros que se le debe pasar son:

- **<filename>:** Este es el nombre del documento donde se guardará la malla en formato STL.

- **<meshName>:** Nombre de la malla, no utilizado en el simulador.
- **<width> , <length>:** Anchura y longitud, en metros, de la malla rectangular.
- **<wDivisions>, <lDivisions>:** Número de divisiones de la malla en la dirección de la anchura y de la longitud.
- **<xcm>, <ycm>, <zcm>:** Posición del centro de masas de la malla rectangular.

Un ejemplo de llamada al programa es el siguiente:

```
./fabricCreator environment/bodies/dpo_clothDoc cloth 1 1 15 15 0.4
0.3 1.3
```

Llamando al programa con estos parámetros, desde la carpeta raíz del programa, genera un documento llamado *dpo_clothDoc* en el directorio *environment/bodies* que contiene la información, en formato STL, de un objeto llamado *cloth*. Éste tiene una malla cuadrada de 1x1 m^2 de 15 divisiones en los dos sentidos y el centro de gravedad en la posición [0,4 0,3 1,3].

Aunque útil, sobretodo en los primeros estadios del desarrollo de *dpoSimulator*, la capacidad de *fabricCreator* para crear geometría está muy limitada. Por esto, a continuación se explica como importar mallas más complejas desde CATIA.

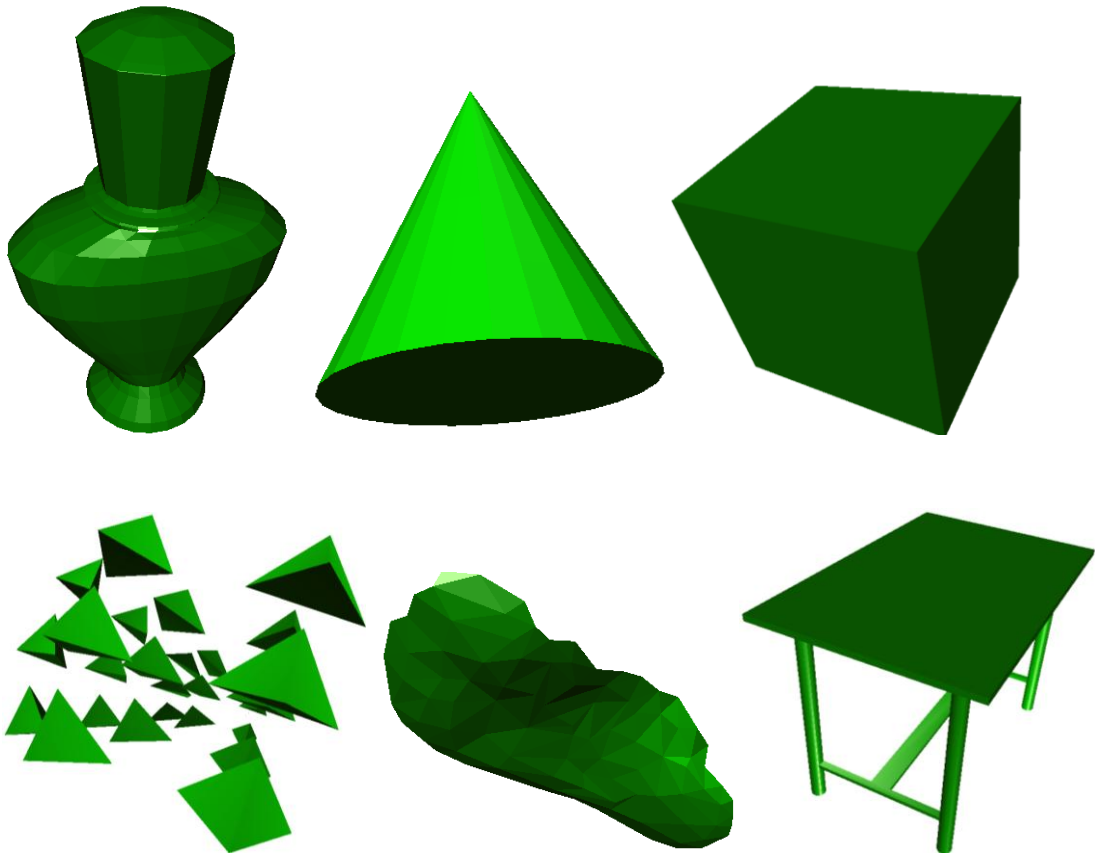


Figura 2.72: DpoSimulator permite la importación de cualquier malla triangular en formato STL.

Generación e importación de mallas desde CATIA:

Los archivos en formato STL, que contienen información sobre mallas que componen la escena, se pueden generar en CATIA V5 siguiendo los pasos que se describen a continuación:

1. El primer paso consiste en generar la geometría en CATIA V5. Ésta puede ser de dos tipos:

    a. **Objetos sólidos:** Los objetos sólidos se pueden crear, por ejemplo, en el workbench de CATIA V5 "*Part Design*". Normalmente, los objetos rígidos que forman parte de una escena, simulada en *dpoSimulator*, se crearán aquí. Se omiten detalles sobre el diseño y creación de geometría en CATIA V5, para más información se remite el lector a la extensa documentación del programa. Los objetos sólidos se pueden guardar fácilmente en formato STL, por lo que el usuario puede pasar directamente al paso 4.

    b. **Superficies:** Interesa utilizar superficies para representar objetos planos deformables. Esto se debe a que representarán la fibra neutra del tejido, para más información consulte el apartado 2.2.2. Además, el workbench de generación de superficies en CATIA V5, *"Generative Shape Design"*, permite generar superficies de elevada complejidad. El inconveniente de las superficies, es que no se pueden guardar directamente en formato STL, teniendo que pasar por los pasos 2 y 3 primero.

    La Figura 2.73 muestra una camiseta representada en una sola superficie. Si ésta se guardase ahora en formato STL, el archivo estaría vacío de información geométrica.

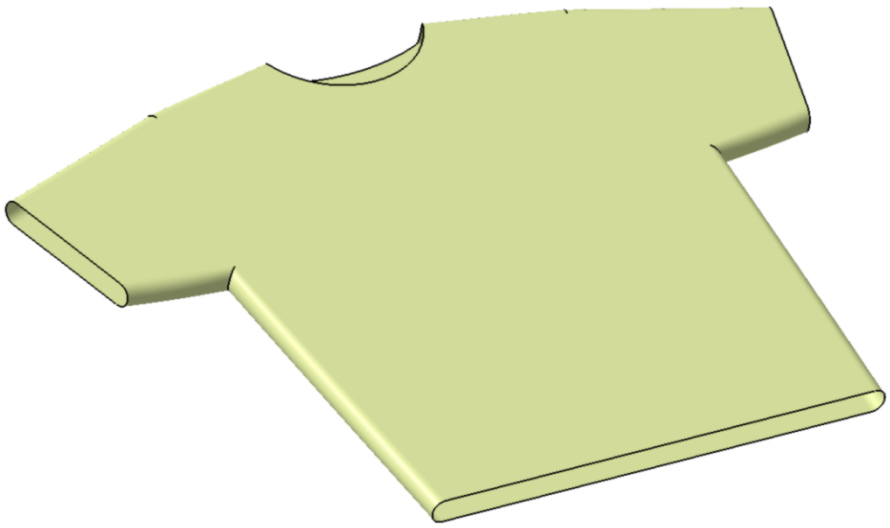


Figura 2.73: Superficie que representa una camiseta.

2. Aunque una superficie no se puede guardar, directamente, en formato STL, sí se puede mallar y guardar la malla. Se puede mallar la superficie con la herramienta *Tessellation*, disponible en el workbench *Machining/STL Rapid Prototyping*. Conviene marcar *Step* para controlar el valor máximo de la longitud de las aristas de los triángulos, ver la imagen a la izquierda en la Figura 2.74, buscando que la malla sea regular. Una malla lo más regular posible favorecerá los cálculos numéricos que realiza el simulador *dpoSimulator*. El mallado de la camisa da como resultado la malla mostrada en la imagen derecha de la Figura 2.74. Se observa que, aunque la malla se adapta muy bien a la superficie original, hay aristas que son muy pequeñas, lo cual hace que la malla sea irregular. Se debe activar la vista inalámbrica, *Wireframe (NHR)*, en CATIA V5 para poder ver la malla.

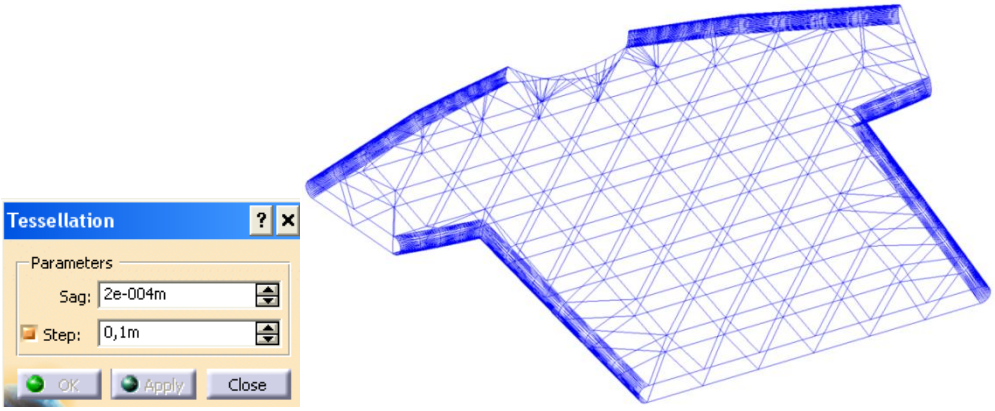


Figura 2.74: Izquierda: Cuadro de diálogo de la herramienta de teselación. Derecha:Camiseta teselada.

3. Para hacer que la camisa sea más regular, se puede utilizar la herramienta *Decimate*, también en el workbench *Machining/STL Rapid Prototyping*. En ésta se activará la opción *Edge Length* y un *Minimum* del mismo valor, o parecido, al asignado en el caso anterior a *Step*. De esta forma, se limitará el valor mínimo de la longitud de las aristas a un valor cercano al máximo. El resultado es que la malla es más regular a costa de no representar tan bien a la superficie original, como se puede ver en la Figura 2.75.

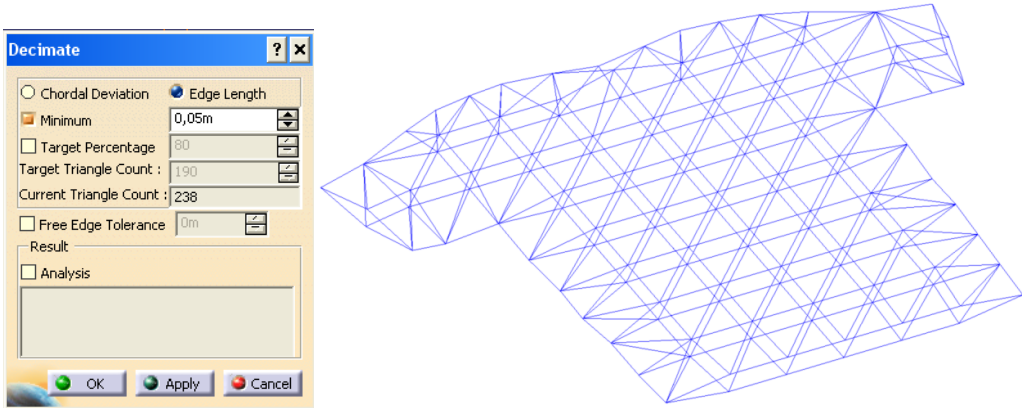


Figura 2.75: Izquierda: Cuadro de diálogo de la herramienta Decimate. Derecha: Malla de la camisa decimada y completamente regular.

4. El último paso consiste en guardar el objeto, ya sea un sólido o una malla, en formato STL. Para ello, simplemente se realiza la acción *File/Save As…*, que abre el cuadro que se muestra en la Figura 2.76. Cabe destacar que, en el caso de guardar un sólido, el programa generará automáticamente una malla para guardar en formato STL. Esta malla probablemente sea muy irregular, como se puede ver en la mesa de la Figura 2.53. Sin embargo, esto poco importa pues la mesa, al ser un objeto rígido, no aparece en los cálculos numéricos, por lo menos de forma directa.

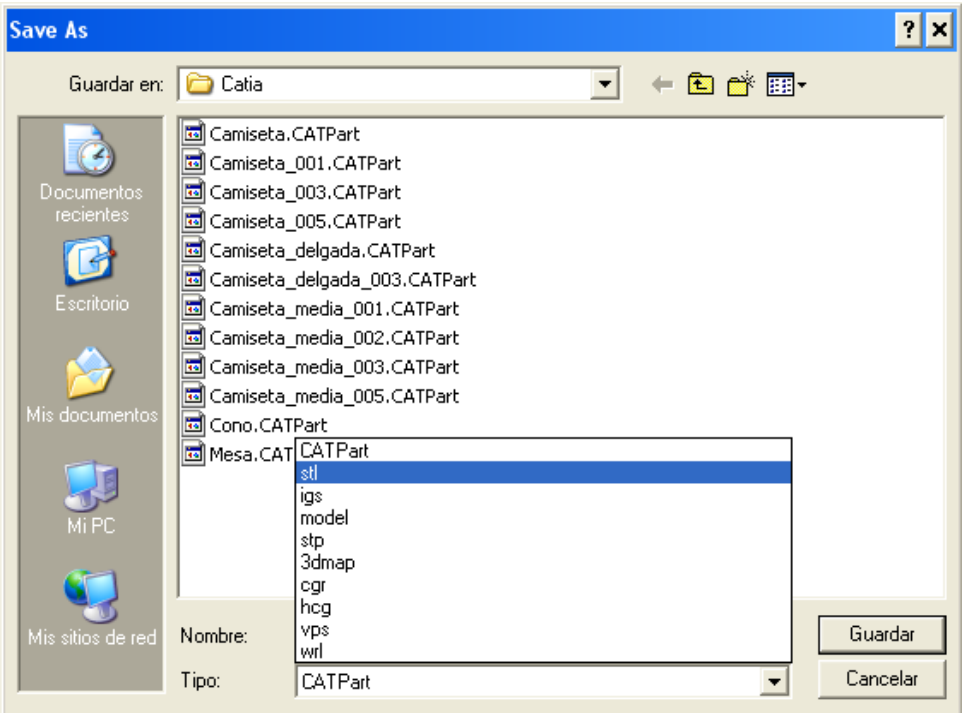


Figura 2.76: CATIA V5 permite guardar una malla en formato STL.

Configuración de los parámetros de los objetos

Hay varios parámetros de los objetos, tanto los rígidos como los deformables, que se pueden modificar en los documentos prefijados con *par* que se encuentran en el directorio *environment/bodies*. Estos documentos son parecidos al mostrado a continuación:

```
The user can enter here the parameters of each body. The parameters
should
not be reordered or deleted. Rigid and deformable objects share
some parameters.
Unecessary parameters will not be read:

Density (kg/m^2 only used for DPO) = 1
Mass (kg only used for Rigid Objects) = 0
Thickness (m)(for Rigid Objects this should be zero) = 0.003
Diffuse color (rgb) =  0 0 0.9
```

El usuario puede modificar tres parámetros fundamentales para el objeto al que se refiere el documento:

- **Densidad o masa:** Para objetos deformables se definirá la densidad y para objetos rígidos se definirá la masa. En cualquier caso, los dos parámetros deben aparecer, simplemente ocurrirá que, según el tipo de objeto, uno de los parámetros será omitido por *dpoSimulator*. En el estado actual del simulador, se supone que los objetos rígidos son infinitamente pesados, para más información consulte el apartado 2.2.2, lo que se representa con una masa nula. No se puede saber, a priori, a qué tipo de objeto se refiere el documento de parámetros mostrado más arriba. *DpoSimulator* determinará el tipo de objeto en función del prefijo del documento de la malla de dicho objeto. Si este es un objeto plano deformable, tendrá una densidad de 1 kg/m^2, y si es un objeto rígido tendrá una masa infinita.
- **Grosor:** Sólo las mallas de objetos deformables planos tienen un grosor, pues representan la fibra neutra, para más información consulte el apartado 2.2.2. Por ello, el grosor para

objetos rígidos debería ser nulo. En el caso mostrado más arriba, el grosor es de 3 mm, por lo que se trata de un objeto plano deformable, *dpo*.

- **Color difuso:** Este parámetro indica de qué color, en formato RGB, se mostrará el objeto en el simulador. En el caso mostrado más arriba, éste es de color azul.

Definición de las restricciones de velocidad

Las restricciones de velocidad permiten al usuario manipular los objetos de la escena. Por ejemplo, aplicando una restricción de velocidad [0 0 0] a los nodos de las dos esquinas de un tejido con malla cuadrada, se obtiene como resultado la escena *HangingCloth2*, en el que un trapo cuelga de sus dos esquinas, ver la Figura 2.69.

Estas restricciones se definen para cada objeto, sea sólido o deformable, en su documento con prefijo *vel*, por ejemplo *vel_HangingCloth2*. Este documento, para un objeto deformable, debe tener un formato parecido al siguiente:

```
Time 0
Number of constraint vectors 1
VertexId 0 Vector [ 0 0 0 ]

Time 1
Number of constraint vectors 1
VertexId 0 Vector [ 0 0 0.1 ]

Time 2
Number of constraint vectors 0

Time 2.5
Number of constraint vectors 2
VertexId 0 Vector [ -0.1 0.1 0 ]
VertexId 10 Vector [ -0.1 0.1 0 ]
```

Este ejemplo de documento *vel* representa las siguientes restricciones de velocidad:

- De 0 a 1 segundos, el nodo 0 estará quieto.
- De 1 a 2 segundos, el nodo tendrá una velocidad de 0,1 en la dirección *z*, y una velocidad nula en las otras direcciones.
- De 2 a 2,5 segundos, no habrá ninguna restricción de velocidad.
- A partir de 2,5 segundos, tanto el nodo 0 como el 10 tendrán una velocidad de [-0,1 0,1 0] .

Evidentemente, el usuario podría ir añadiendo más líneas de forma manual para añadir más restricciones de velocidad.

Los documentos *vel* de objetos rígidos tienen el mismo formato. Pero, con este tipo de objetos, se debe manipular únicamente *un* nodo, para asegurar que se mantiene rígido. Por ahora, el simulador no contempla la rotación de objetos rígidos. En las escenas en las que aparece el objeto mesa, *table*, en el CD adjunto, dicho objeto no se mueve. Por ello, su documento de restricción de velocidades, *vel_table*, es el siguiente:

```
Time 0
Number of constraint vectors 1
VertexId 0 Vector [ 0 0 0 ]
```

Se pueden modificar los documentos de restricciones de velocidades manualmente para obtener trayectorias sencillas. No se ha desarrollado ninguna herramienta genérica que permita crear trayectorias más complejas, y que asegure que no se produzcan sacudidas, pues, como se explica

en la Introducción, el objetivo a largo plazo es que *el programa* escoja que restricciones de velocidad debe aplicar. Pues el objetivo del proyecto de investigación, para el que se ha desarrollado *dpoSimulator*, es crear un programa que determine cómo se debe manipular una pieza de ropa determinada para que pase de una posición a otra.

Aun y así, se ha desarrollado un programa en Matlab, *matlab/TrajectoryGenerator.m*, que genera el documento de restricción de velocidades para el caso concreto del objeto *ShirtOnTable*. Éste genera una trayectoria compleja, con muchos puntos, asegurando que no haya sacudidas. Si se desea, se puede tomar como punto de partida para desarrollar una herramienta de manipulación de objetos más genérica y completa. Un buen comienzo sería la programación de una función que generase trayectorias rectilíneas con descansos en los extremos, para que el perfil de velocidades sea continuo y no haya sacudidas.

2.3.2 Manual de desarrollo

Principales herramientas de programación utilizadas

Tanto el simulador *DpoSimulator* como los programas auxiliares *fabricCreator*, *stlViewer* y *meshMover* se han programado en C++. El lenguaje de programación C++ es una ampliación de C que permite la programación orientada a objetos (Ceballos, 2007). Este tipo de programación se basa en la creación de *objetos* de diferentes *clases*, cada uno con sus propiedades y sus funciones o *métodos*, que interactúan los unos con los otros. Esto permite organizar de forma sencilla programas que, de otra forma, serían muy complejos. En los Anexos se presenta parte de la documentación de las clases del programa y en este apartado se describe en breve el flujo principal del programa y se dan recomendaciones sobre cómo utilizarlo como base para desarrollar nuevas aplicaciones, respectivamente.

El programa se ha desarrollado en el entorno de programación KDevelop 3.5.3 y se recomienda que cualquier ampliación se desarrolle en el mismo entorno. Pues el archivo de proyecto *pmdo.project* que se encuentra el en directorio raíz del programa está configurado para compilarlo y enlazarlo de forma correcta.

Se utiliza la librería Qt, versión 3.3, para programar la interfaz gráfica del panel de control. Ésta facilita la creación cuadros de diálogo, botones, etiquetas y cuadros de texto de forma parecida a Visual Basic. Aunque la apariencia física de la interfaz se puede crear de forma gráfica con el programa QtDesigner, se ha preferido programarla a mano directamente, para tener más control. Esta librería también facilita la programación por eventos con su sistema de *slots* (Trolltech).

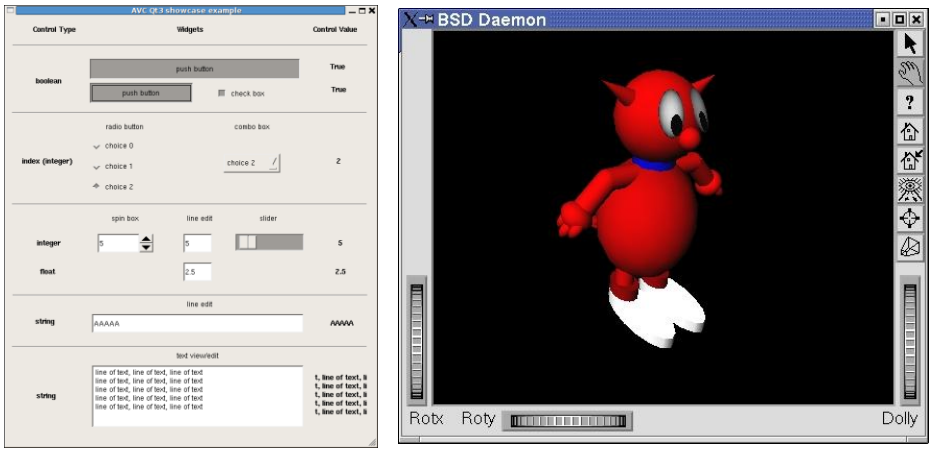


Figura 2.77: Izquierda: Interfaz de usuario programada con la librería Qt3.3. Derecha: Visualizador tridimensional de la librería Coin3d/Open Inventor.

Finalmente, cabe destacar Doxygen como herramienta utilizada para documentar el código. Siguiendo un formato concreto de documentación, Doxygen permite compilarlo en una documentación interactiva en formato html (CISST).

Organización de los directorios

Todos los archivos, exceptuando las bibliotecas listadas en el apartado 2.3.1, necesarios para ejecutar el simulador *dpoSimulator* y sus programas auxiliares se encuentran en el directorio raíz, *pmdo*, del programa. Cuando el programa se compila, los ejecutables aparecen en el directorio raíz. Éste se encuentra en el CD adjunto y está tiene las siguientes carpetas:

- **environment:** Esta carpeta contiene la información de la escena que *dpoSimulator* cargará. Para más información consulte el apartado 2.3.1.
- **environments:** Esta carpeta contiene diferentes carpetas con escenas predeterminadas, mostradas en la la Figura 2.71. Para cargar una escena predeterminada, simplemente se ha de reemplazar la carpeta *environment* por la carpeta de la escena, y cambiarle a esta el nombre a *environment*.
- **frames:** En esta carpeta, *dpoSimulator* guarda las imágenes, o *frame*, de todos los pasos de la simulación. Con estas imágenes después se puede crear un video, apartado 2.3.1.
- **html:** Doxygen guarda en esta carpeta la documentación del programa, al compilarla. La página principal de documentación es *index.html*.
- **iterationMeshes:** *DpoSimulator* guarda aquí las mallas que se producen en cada iteración de la resolución de ecuaciones, ver la Figura 2.70.
- **matlab:** En esta carpeta se halla el archivo *TrajectoryGenerator.m*. Éste permite crear el documento de restricción de velocidades del objeto *ShirtOnTable*, para más información consulte el apartado 2.3.1.
- **src:** Este directorio contiene todo el código de *dpoSimulator* y sus programas auxiliares. Contiene las siguientes carpetas:
    - **fabricCreator:** Aquí está el código fuente de la aplicación *fabricCreator*. Esta permite crear mallas rectangulares sencillas, para más información consulte el apartado 2.3.1.
    - **freecloth:** Aquí está la librería *freecloth* ligeramente modificada para las necesidades del simulador. Esta librería se encarga de la simulación de la dinámica interna de la ropa y es una implementación del algoritmo de Baraff y Witkin (Pritchard, Implementing Barraf & Witkin's Cloth Simulation, 2009). Para más información consulte el apartado 2.2.1.
    - **meshMover:** Aquí se halla el código del programa auxiliar *meshMover*, que permite traslaciones simples mallas y genera documentos STL con las mallas en las nuevas posiciones. De esta forma se puede ajustar con facilidad la posición de las mallas en las escenas.
    - **PMDO:** Aquí se encuentra el código del simulador *dpoSimulator*, el programa principal desarrollado en este proyecto. Parte del programa se encuentra organizado en los siguientes directorios:
        - **PMDO:** El código de cálculo de la simulación, propiamente dicho, está en esta carpeta. Aquí es donde se encuentra la implementación del algoritmo de detección y resolución de contactos y colisiones, algoritmo

descrito en (Bridson, Fedkiw, & Anderson, 2002). Parte de este código está organizado en las siguientes carpetas:

- **AABB:** Todo el código relativo a cajas delimitadoras alineadas con los ejes, *Axis Aligned Bounding Boxes*, está en esta carpeta. Para más información sobre las AABB y su utilidad para la simulación, consulte el apartado 2.2.3.
- **Detection:** Aquí está el código referente a la detección precisa de contactos y colisiones y se realizan llamadas a funciones de la carpeta *AABB* para detecciones aproximadas. En esta carpeta, se implementan los algoritmos descritos en los apartados 2.2.3 y 2.2.4 en lo referente a cajas delimitadoras.
- **Responser:** Los algoritmos de resolución de contactos y colisiones se implementan en este directorio. Para más información sobre estos algoritmos consulte el apartado 2.2.5

▪ **Viewer:** Las clases y funciones que se encargan de la interfaz de usuario y mostrar los resultados se encuentran en esta carpeta. Se podría decir que ésta es la capa superior del programa. Realiza llamadas al simulador, *PMDO*, y muestra los resultados. Para más información, consulte la sección 2.3.2. Cabe destacar el documento CoinGeometryAdder, que contiene un conjunto de funciones que permite, con facilidad, añadir geometría de soporte a la escena (flechas, cubos…). Esta geometría es la que se utiliza para mostrar AABB, contactos y colisiones, ver el apartado 2.3.1.

o **stlViewer:** Aquí se halla el código del programa auxiliar *stlViewer*. Éste permite visualizar de forma sencilla un número elevado de mallas, para más información consulte la página 87.

Descripción general del código

A continuación se describe el flujo del programa durante la simulación. Se entra en detalle en el código y se da una guía que, junto a la documentación adjunta en los Anexos y el CD, permitirá a un desarrollador utilizar el simulador como base para otros proyectos. Se supone que el lector tiene conocimientos sobre la programación orientada a objetos en C++.

Construcción de los objetos del simulador:

La función principal *main*, se encarga de construir el visualizador del mundo, *worldViewer*, se llama mundo al conjunto de todos los objetos. En */src/PMDO/Viewer/main.cpp* está la función *main*:

```
int main ( int argc, char ** argv )
{
 QApplication app ( argc, argv );
 WorldViewer worldViewer ( 0 , "worldViewer" );
 return 0;
}
```

WorldViewer es una clase que hereda de la clase *QObject*, de *Qt*, y se encarga de construir todos los objetos necesarios para la simulación. El constructor de *WorldViewer*, en *WorldViewer.cpp*, crea el mundo (*world)*, la interfaz de visualización de la escena *ExplorerWidget*, el panel de control *ControllerWidget* e inicia el bucle principal.

```
WorldViewer::WorldViewer ( QObject * parent, const char * name)
:QObject ( parent, name ),_world ( new World() ),_stepFlag ( false
)
{
    ...
    _explorerWidget = new ExplorerWidget ( 0 , _world );
    _controllerWidget = new ControllerWidget ( 0 , _world);
    SoQt::mainLoop();
    ...
}
```

Se observa que el bucle principal es de *SoQt*, que permite ejecutar simultáneamente *Qt* y *Coin* de forma sencilla. El *explorerWidget* se encarga de un objeto *worldScene* que se encarga de representar gráficamente, en la ventana de Open Inventor, la escena que se está simulando, para ello busca las mallas en *world*. El objeto de la clase *World* no se encuentra en la carpeta *src/Viewer* sino en la *src/PMDO*, pues ya es una clase *esencial* para la simulación y no solo de visualización. La homóloga de la clase *World* pero para visualización es *WorldScene*, que se encuentra junto con las otras clases de visualización en *src/Viewer*.

El constructor de *World*, en *World.cpp*, se encarga de cargar los parámetros del mundo, *_params*, y los objetos que componen la escena, *_objects*, los cuales pueden ser de tipo rígido, *RigidObject*, o deformable plano, *DeformablePlanarObject*. Tanto los parámetros como los objetos se cargan de los archivos en la carpeta *environment*. Por último, el constructor del mundo se inicia con tiempo cero, *_time*, y se crea el objeto *_stepper*. Éste se encargará más adelante de avanzar el mundo un paso tras otro, aplicando una estrategia que acelere la simulación.

```
World::World(): _objects ( new WorldObjects ( this ) ), _time ( 0 )
{
    //Load parameters from parameters file:
    _params.loadFromFile();

    //Load objects from the files in environment/bodies:
    _objects->loadFromFiles();

    _time = 0;
_stepper =RCShdPtr<World::WorldStepStrategy>(new
 World::WorldStepStrategy(this));
}
```

Ejecución de la simulación:

Una vez se ejecuta el bucle principal, el programa espera a que se produzca un evento. Si este es de visualización, por ejemplo mostrar los árboles de cajas delimitadoras o los contactos detectados, se llaman las funciones del archivo *WorldScene.cpp*. En caso de que sea un comando de ejecución de la simulación, *play o step*, se llama a la función *stepTimer* en *WorldViewer.cpp*. Ésta le dice al mundo, *world*, que dé un paso de simulación y después actualiza la escena que se muestra:

```
static void stepTimer ( void *data, SoSensor * )
{

    //Obtain data, which is the world
    WorldViewer * theData = ( WorldViewer * ) data;

    //Step foward in time.
    World * world = theData->getWorldPtr();
    world->step();

    //Update rendering models.
    theData->sendUpdateSignal();
```

```
        //Update displayed time.
        theData->emitUpdateTimeDisplay();

    //If this is a single step then stop loop, in any other case the
    //simulation will continue.
        if ( theData->getStepFlag() )
        {
       WorldViewer::TimeCallback * timeCallback = theData-
       >getTimeCallback();
            SoTimerSensor& timerSensor = timeCallback->getSensor();
            timerSensor.unschedule();
        }
    }
```

La llamada al método *step* de *world*, llama, a su vez, al *step* de *WorldStepStrategy*, en *WorldStepStrategy.cpp*. Este se encargará de dar los pasitos necesarios para ejecutar un paso de la simulación. Para ello, primero prepara los parámetros, *preColsteps()*, luego va ejecutando pasos de colisiones hasta que se cubre todo el paso, *colStep()*, y luego realiza un postprocesado *postColSteps()*:

```
void World::WorldStepStrategy::step()
{
    //The data members are prepared:
    preColSteps();

//We take collision steps till the whole lapse of time between
//frames is covered.
    while ( ! colStepsDone() )
    {
        colStep();
    }

    postColSteps();

}
```

A continuación se muestra una versión simplificada del método *colStep()*. Ésta tiene como objetivo ilustrar la estrategia de resolución de contactos y colisiones y relacionar los algoritmos descritos en el apartado 2.1 con las clases y funciones que los implementan:

```
void World::WorldStepStrategy::colStep()
{
 //Create a copy of the objects at time n and advance them,
 //allowing collisions, to time n+1:
 RCShdPtr<WorldObjects> proposedObjects ( new WorldObjects (
 *worldObjects ) );
 proposedObjects->stepAllowingCollisions ( _colH );

 //The proposed mean velocities of all the objects in world are
 //calculated.
 RCShdPtr<WorldMeanVelocities> worldMeanVelocities ( new
 WorldMeanVelocities( worldObjects, proposedObjects,
 BaTime::durationAsSeconds ( _colH ) ) );

 //Detect the contacts between the objects in the world and
 //modify the mean velocities accordingly
 WorldProximityDetector worldProximityDetector ( worldObjects,
 worldMeanVelocities );
 WorldProximityResponser worldProximityResponser (
 worldMeanVelocities, worldProximityDetector, PARALLEL, IMPLICIT );
```

```
        //Now collisions are calculated from the initial position
        //(worldObjects) and the worldMeanVelocities which have been
        //modified by contact impulses.
        WorldCollisionDetector worldCollisionDetector ( worldObjects,
        worldMeanVelocities );
        //Number of detected collisions
        int nCollisions = worldCollisionDetector.getPTs().size() +
        worldCollisionDetector.getEEs().size();

        if ( nCollisions != 0 ){

              //Collisions have been detected so we check if the collision
              step //is small enough to proceed with a collison response.
        if ( isColHSmallEnough() )
              {
                    //Solve collisions iteratively untill all collisions
                    //untill a new collision free state is reached.
                    solveCollisionsIteratively ( worldObjects,
                    proposedObjects, worldMeanVelocities,
                    &worldCollisionDetector );
                    //Update time
                    _time += _colH;
              }else{
                    //The step is not small enough, so we must repeat the
                    //collision step with a smaller _colH. _colH is
                    //halved to //proceed with a smaller collision step.
                    halveColH();
              }
        }else{
        //No collisions have been detected
        if(nContacts != 0)
               {
                    //If contacts have been detected, proposedObjects is
                    //updated to account for them.
                    worldProximityResponser.applyMutatedStepTo (
                    *worldObjects,  *proposedObjects );
               }

              //There were neither contacts not collisions so the proposal
              //(modified by contacts or not) is accepted.
               _world->setObjectsPtr ( proposedObjects );
        //Update time
               _time += _colH;
        }
      }
```

Se podrían enumerar los siguientes pasos:

1. Copia de los objetos en el estado *n* y simulación de un paso para estos objetos teniendo solo en cuenta la dinámica interna, se obtiene *proposedObjects*.
2. Cálculo de las velocidades medias de todos los objetos del mundo, *worldMeanVelocities*, desde su posición inicial, *worldObjects*, y su posición estimada para el final del paso, *proposedObjects*.
3. Detección de contactos para *worldObjects*, según la posición inicial de los objetos, y modificación de *worldMeanVelocities* de acuerdo con estos contactos. Aquí se aplica la resolución de contactos por impulsos descrita en el apartado 2.2.5.
4. Detección de colisiones entre objetos, partiendo de *worldObjects* y con las velocidades medias *worldMeanVelocities*. Aquí hay dos posibilidades:

a. **Si se detectan colisiones:** En caso de que el paso de colisión actual sea lo suficientemente pequeño, se resuelven las colisiones de forma iterativa, con la función *solveCollisionsIteratively*. En caso contrario, se repite el procedimiento desde el paso 1 pero con un paso la mitad de grande.

b. **Si no se detectan colisiones:** Se modifica *proposedObjects* según los contactos calculados, en caso de que los haya, y se asigna como estado al final del paso.

La función *colStep* se llamará el número necesario de veces para que se cubra todo el paso de simulación. Cabe destacar que es diferente el *paso* de simulación, cuyo periodo define el usuario en el documento *environment/world_parameters/world_parameters/*, que el *paso de colisión*. Éste último es modificado de forma automática por *World::WorldStepStrategy*, para asegurar que la resolución de colisiones se hace únicamente para intervalos pequeños. Pues, como se indica en el paso 4a, las colisiones sólo se resuelven si el paso es lo suficientemente pequeño (Bridson, Fedkiw, & Anderson, 2002). De esta forma, desde *World::WorldStepStrategy*, y en concreto desde su método *colStep*, se puede controlar cómo se resuelven los contactos y las colisiones para que la simulación se lleve a cabo de forma satisfactoria.

Recomendaciones

- El desarrollador puede utilizar las clases de simulación, situadas en el directorio *src/PMDO*, con otra interfaz, es decir sin usar *Viewer*. Para ello, se recomienda el uso directo de la clase *World*. Ésta carga la escena de la carpeta *environment* y permite simularla con el método *step*.
- La simulación, en el estado actual, no permite retroceder un paso la simulación. Para ello, la mejor opción sería copiar el mundo y guardarlo. Al rebobinar, sólo se tendría que mostrar el mundo, en un estado anterior, que se requiera.
- No es necesario que el desarrollador entre en capas que estén en una capa inferior que *World*, pues ésta tiene todas las herramientas necesarias para simular. La única excepción son las clases *DeformablePlanarObject* y *RigidObject*, pues estas están diseñadas para cargar las restricciones de velocidad de un archivo, *velConstraintFileName*:

```
///Constructor of the deformable planar object class.
DeformablePlanarObject ( int fps, float gravity,
RCShdPtr<GeMesh> initialMesh, const String& parFilename,
const String& velConstraintFileName);

///Constructor of the rigid object class
RigidObject ( const String& meshFilename, const String&
parFilename, const String& velConstraintsFilename );
```

  Un planificador de la manipulación de tejidos, para el que se ha desarrollado *dpoSimulator*, cómo se explica en la Introducción, requerirá tener un control directo sobre las restricciones de velocidad de los objetos. Lo que es más, el planificador será el que escogerá qué restricciones de velocidad se deben aplicar. Por ello, no será oportuno que las velocidades se carguen de un archivo, por lo que los constructores de estas dos clases se deberán cambiar.
- Por restricción de tiempo, no se ha acabado de optimizar el código. Se cree que un programador experimentado puede optimizar más el programa, en caso de que sea necesario, utilizando un *profiler*. Éste indica el porcentaje de tiempo que se dedica a cada función de forma acumulativa. El Diagrama 2.15 muestra de forma gráfica el resultado del profiler *Valgrind*. Para utilizarlo, se escribe la siguiente instrucción en una ventana de terminal en el directorio raíz del programa:

```
valgrind --tool=callgrind --instr-atstart=no ./simulator
```

Esta instrucción ejecutará el programa y empezará a monitorizar el consumo de cada función. Seguidamente, se debe ejecutar la simulación hasta que se considere que ésta es una muestra representativa del conjunto de simulaciones que se puede realizar, es decir que hay contactos y colisiones pero no en exceso. Para visualizar los resultados, como en el Diagrama 2.15, se escribe la siguiente instrucción en otra terminal en el directorio raíz del programa:

```
callgrind_control -i on <PID>
```

Dónde PID es el número de identificación del proceso, *dpoSimulator*, en Linux. Éste se puede saber con la herramienta *top* o *htop*.

Se observa, en el Diagrama 2.15, que la simulación de la dinámica interna, *stepAllowingCollisions*, consume un buen porcentaje de la simulación. Puesto que la dinámica interna viene implementada por *freecloth*, no se ha modificado apenas, pero se podría hacer si se considerase necesario para optimizar el programa.

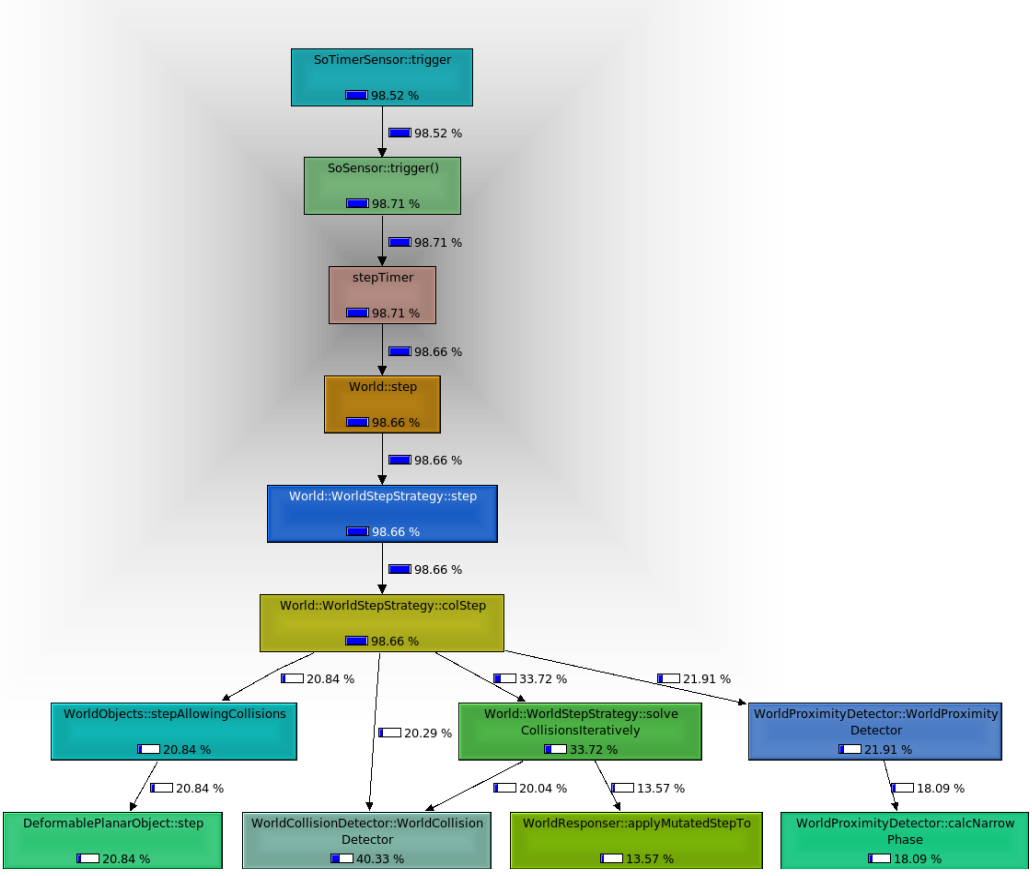


Diagrama 2.15: El cálculo de la dinámica interna de los tejidos, stepAllowingCollisions, requiere un gran porcentaje del tiempo de simulación, en este caso un 21%.

3 Conclusiones

En este proyecto final de carrera se ha desarrollado un simulador de tejidos, *dpoSimulator*, que tiene en cuenta los contactos y colisiones de los objetos planos deformables, tejidos, consigo mismos y con otros objetos, ya sean otros tejidos u objetos rígidos, como se muestra en la Figura 3.1. El desarrollo se ha basado en la implementación del algoritmo descrito en (Bridson, Fedkiw, & Anderson, 2002). El simulador tiene las siguientes características:

- Puede cargar cualquier escena, con objetos y parámetros definidos por el usuario, pues puede importar mallas en formato STL, cómo se indica en el apartado 2.3.1. Se puede cambiar fácilmente la escena a mostrar cambiando la carpeta *environment*.
- Se puede manipular los objetos aplicando restricciones de velocidad, para más información consulte la sección 2.3.1.
- Ofrece la posibilidad de visualizar los contactos y colisiones detectados, árboles de cajas delimitadoras y guardar la escena cómo imágenes.
- Tiene todas las ventajas de un entorno 3D. Permite moverse por las escenas y mostrarlas con diferentes estilos de visualización.

También se han desarrollado un conjunto de aplicaciones auxiliares:

- *stlViewer*: Permite cargar con facilidad un conjunto de mallas en formato STL en un entorno 3D.
- *meshMover*: Permite desplazar todos los nodos de una malla en formato STL, cambiando así su posición en la escena.
- *fabricCreator*: Permite crear con facilidad mallas planas rectangulares de diferentes densidades y guardarlas en formato STL.

El simulador cumple con las características requeridas, planteadas en el apartado 1.2.

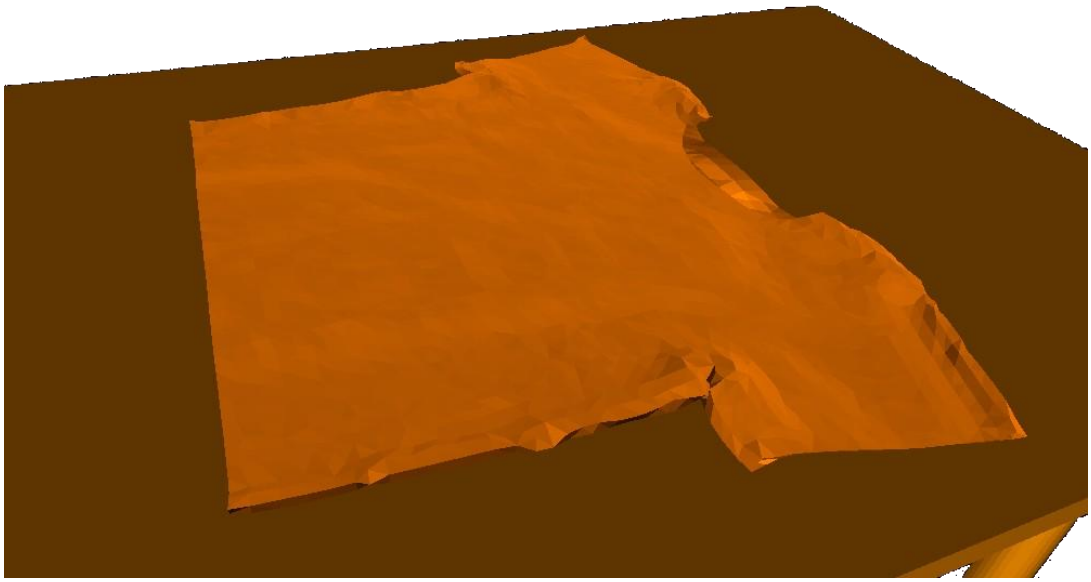


Figura 3.1: Simulación de la escena ShirtOnTable en dpoSimulator.

3.1 Recomendaciones

Aunque el simulador cumple con las especificaciones requeridas, se pueden desarrollar las siguientes ampliaciones:

- Cómo se indica en el apartado 2.3.1, no se ha desarrollado ninguna herramienta que facilite la generación de trayectorias complicadas y que las traduzca a restricciones de velocidades. Esto se debe a que será el planificador el que *decida* cómo manipular los objetos. Sin embargo, en el documento *matlab/TrajectoryGenerator.m* puede servir de

punto de partida para desarrollar un generador de trayectorias automático de forma sencilla.

- Actualmente, sólo se puede *trasladar* los objetos rígidos, aplicando una restricción de velocidad a uno de sus nodos, pero no se puede *rotar*. Esto es una limitación que se puede solucionar con facilidad cambiando el documento de restricciones de velocidades para objetos rígidos para que acepte rotaciones, por ejemplo.
- La simulación de la dinámica interna de los tejidos, según (Baraff & Witkin, Large Steps in Cloth Simulation, 1998), depende de unos parámetros, resistencia a tracción, flexión y cizalla, que caracterizan el mismo. Éstos, diferencian el algodón del cuero, con una mayor resistencia a flexión y cizalla. *DpoSimulator*, en su estado actual, utiliza unos parámetros concretos, que el usuario no puede cambiar. Se puede ampliar el programa añadiendo estos parámetros a la lista de parámetros de los objetos planos deformables. Después, se debería modificar el constructor de la clase *DeformablePlanarObject* para que cargue los parámetros del archivo.
- Por último, hay dos funcionalidades, descritas en (Bridson, Fedkiw, & Anderson, 2002), que no se han implementado por un tema de prioridad y restricción de tiempo:
    - No se han implementado la resolución de colisiones por *zonas rígidas de impacto*. Este algoritmo se utiliza cuando el método de resolución de colisiones estándar no ha conseguido resolver todas las colisiones en un cierto número de iteraciones. En tal caso, se define una zona de impacto que se tratará como si fuese un objeto rígido y se resuelve la colisión para ese objeto rígido. En caso de que se quiera desarrollar este método de resolución de colisiones, se recomienda empezar por el método *WorldStepStrategy::colStep()*, explicado en el apartado 2.3.2. Pues será en éste donde, al cabo de unas iteraciones, se llamará al método de resolución por zonas rígidas de impacto.
    - Tampoco se han implementado las fricciones, pues no han sido una prioridad por dos razones. La primera es porqué las mallas que se utilizan son lo suficientemente bastas como para que, para la mayoría de simulaciones, la fricción pasase desapercibida. La segunda es la complejidad de la organización de los coeficientes de fricción estáticos y dinámicos entre los diferentes objetos de la escena. En cualquier caso, la fricción, en caso de implementarse, actuaría en forma de impulsos que se sumarían a los de repulsión debido al contacto (Bridson, Fedkiw, & Anderson, 2002).

4 Bibliografía

1. Andrade-Cetto, J. (2005). *The Kalman Filter.* Barcelona.
2. Ascher, U. M., & Boxerman, E. (n.d.). On the modified conjugate gradient method in cloth simulation. *Visual Computer manuscript* .
3. Baraff, D., & Witkin, A. (1998). Large Steps in Cloth Simulation. *Computer Graphics Proceedings, Annual Conference Series* , 43-54.
4. Baraff, D., Witkin, A., & Kass, M. (2003). Untangling Cloth. *Proceedings of ACM SIGGRAPH 2003 , 22* (3), 862-870.
5. Bradski, G., & Kaehler, A. (2008). Histograms and Matching. In G. Bradski, & A. Kaehler, *Learning OpenCV Computer Vision with the OpenCV Library* (pp. 193-221). Sebastopol: O'Reilly Media.
6. Bridson, R., Fedkiw, R., & Anderson, J. (2002). Robust treatment of collisions, contact and friction for cloth animation. *ACM Transactions on Graphics* , vol. 21, no. 3.
7. Ceballos, F. J. (2007). *Programación orientada a objetos con C++* (4ª Edición ed.). Madrid: RA-MA.
8. CISST. (n.d.). *How to document code for Doxygen*. Retrieved 08 25, 2008, from http://www.cisst.org/cisst/cis/doxygen-documentation.html
9. Cuén-Rochín, S., Andrade-Cetto, J., & Torras, C. *Action Selection for Robotic Manipulation of Deformable Planar Objects.*
10. H. Press, W., A. Teukolsky, S., T. Vetterling, W., & P. Flannery, B. (1995). *Numerical Recipes in C* (Segunda edición ed.). Cambridge, Estados Unidos: Cambridge University Press.
11. Lowe, D. G. (2004). Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision* .
12. Pritchard, D. (2003, 04 25). Retrieved 08 21, 2010, from freecloth project: http://davidpritchard.org/freecloth/
13. Pritchard, D. (2009). *Implementing Barraf & Witkin's Cloth Simulation.* Retrieved 08 21, 2010, from davidpritchard.org: davidpritchard.org/freecloth/docs/report.pdf
14. Provot, X. (2004). Collision and self-collision handling in cloth model dedicated to design garments. *IEEE Transactions on Visualization and Computer Graphics , 10*, 649-663.
15. Provot, X. (1995). Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth Behavior. *Proceedings in Graphics Interface 1995* , 147-154.
16. *The Inventor Mentor: Programming ObjectOriented 3D Graphics with Open InventorTM, Release 2.* (n.d.). Retrieved 08 25, 2010, from webdocs.cs.ualberta.ca/~graphics/books/mentor.pdf
17. Thrun, S., Burgard, W., & Fox, D. (2006). Planning and Control. In S. Thrun, W. Burgard, & D. Fox, *Probabilistic Robotics* (pp. 485-604). London, England: The MIT Press.
18. Trolltech. (n.d.). Retrieved 08 21, 2010, from Qt Reference Documentation (Open Source Edition): http://doc.trolltech.com/3.3/index.html

19. Volino, P., & Thalmann, N. M. (2000). Implementing Fast Cloth Simulation with Collision Response. *Computer Graphics International, 2000. Proceedings* (pp. 257-266). Geneva: MIRALab.

5 Anexos

5.1 Contenido del CD

El CD adjunto a este documento contiene una copia de la memoria del proyecto y el simulador *dpoSimulator*, con su correspondiente documentación, en la carpeta *pmdo*.

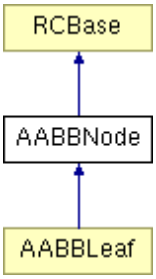
5.2 Documentación de las clases

5.2.1 AABBNode Class Reference

AABBNodes are the nodes of an Axis Aligned Bounding Box Tree.

`#include <AABBNode.h>`

Inheritance diagram for AABBNode:



Public Member Functions

AABBNode (const AABBNode *const parent, const std::vector< AABBContainable * > &elementList, unsigned int depth, unsigned int minElements)

Constructor to build the node and recursively build its children.

AABBNode (const AABBNode *const parent, const std::vector< boost::shared_ptr< AABBContainable > > &elementList, unsigned int depth, unsigned int minElements)

Constructor to build the node and recursively build its children.

~AABBNode ()

Detstructor.

unsigned int getDepth () const

_depth accessor.

const BBox & getBBox () const

_box accessor.

Friends

class TreeBuilder
class AABBTree

Classes

class **TreeBuilder**

Class that is used to group methods regarding tree construction.

Detailed Description

AABBNodes are the nodes of an Axis Aligned Bounding Box Tree.

This class is meant to be used mainly by *AABBTree* . The class can do certain operations recursively: it can build itself recursively and find overlapping leaves recursively. Most functions are meant to be used by *AABBTree* so the user is forwarded to its documentation.

Each node object has a bounding box (*BBox*), which contains with the minimum possible axis aligned box all the elements (*AABBContainable*) that are supplied in its constructor.

See also:

AABBTree, AABBLeaf, BBox, AABBContainable

Constructor & Destructor Documentation

AABBNode::AABBNode (const AABBNode *const *parent*, const std::vector< AABBContainable * > & *elementList*, unsigned int *depth*, unsigned int *minElements*)

Constructor to build the node and recursively build its children.

Parameters:

parent Parent node of this node. In case this node is root parent should be 0.
elementList Elements which the AABBNode's *BBox* must bound.
depth Depth of the node in the tree. The root is depth 0.
minElements Minimum number of elements (*AABBContainable*) that nodes (this one and its descendants) can have.

AABBNode::AABBNode (const AABBNode *const *parent*, const std::vector< boost::shared_ptr< AABBContainable > > & *elementList*, unsigned int *depth*, unsigned int *minElements*)

Constructor to build the node and recursively build its children.

Parameters:

parent Parent node of this node. In case this node is root parent should be 0.
elementList Elements which the AABBNode's *BBox* must bound.
depth Depth of the node in the tree. The root is depth 0.
minElements Minimum number of elements (*AABBContainable*) that nodes (this one and its descendants) can have.

AABBNode::~AABBNode ()

Detstructor.

Member Function Documentation

unsigned int AABBNode::getDepth () const

_depth accessor.

const BBox & AABBNode::getBBox () const

_box accessor.

Friends And Related Function Documentation

friend class TreeBuilder `[friend]`

friend class AABBTree `[friend]`

The documentation for this class was generated from the following files:

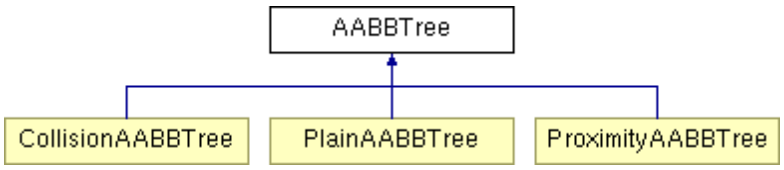
src/PMDO/PMDO/AABB/AABBNode.h
src/PMDO/PMDO/AABB/AABBNode.cpp
src/PMDO/PMDO/AABB/AABBNode.templates.h

5.2.2 AABBTree Class Reference

Class that builds and contains that information of an AABB tree.

`#include <AABBTree.h>`

Inheritance diagram for AABBTree:



Public Member Functions

- AABBTree (const std::vector< AABBContainable * > &elementList, unsigned int minFaces=1)
  Constructor of the class.
- AABBTree (const std::vector< boost::shared_ptr< AABBContainable > > &elementList, unsigned int minFaces=1)
  Constructor of the class.
- void collectBBoxes (std::vector< const BBox * > &bBoxCollection) const
  Class to collect all the bounding boxes of the tree in a vector.
- void collectNodes (std::vector< const AABBNode * > &nodeCollection) const
  Class to collect all the nodes of the tree in a vector.
- void collectOverlappingLeaves (const AABBTree *otherTree, std::vector< FacePair * > &overlappingFacePairs) const
  Class that determines all the leaves, terminal nodes, of this tree that are in contact with another tree.
- void collectOverlappingLeaves (const AABBTree *otherTree, boost::ptr_vector< FacePair > &overlappingFacePairs) const
  Class that determines all the leaves, terminal nodes, of this tree that are in contact with another tree.
- void collectSelfOverlappingLeaves (std::vector< FacePair * > &overlappingFacePairs) const
  Class that determines all the leaves, terminal nodes, of this tree that overlap.
- void collectSelfOverlappingLeaves (boost::ptr_vector< FacePair > &overlappingFacePairs) const
  Class that determines all the leaves, terminal nodes, of this tree that overlap.

Detailed Description

Class that builds and contains that information of an AABB tree.

AABBTree can also return the Bounding Boxes as a vector and detect overlaping boxes.

Constructor & Destructor Documentation

AABBTree::AABBTree (const std::vector< AABBContainable * > & *elementList*, unsigned int *minFaces* = 1)

Constructor of the class.

The constructor recursively builds the tree from its root.

Parameters:

elementList List of objects that can be contained by the AABB nodes. For contacts these are triangles with thickness and for collisions these are triangle pairs.
minFaces Minimum number of elements that a node can contain.

AABBTree::AABBTree (const std::vector< boost::shared_ptr< AABBContainable > > & *elementList*, unsigned int *minFaces* = 1)

Constructor of the class.

The constructor recursively builds the tree from its root.

Parameters:

elementList List of objects that can be contained by the AABB nodes. For contacts these are triangles with thickness and for collisions these are triangle pairs.
minFaces Minimum number of elements that a node can contain.

Member Function Documentation

void AABBTree::collectBBoxes (std::vector< const BBox * > & *bBoxCollection*) const

Class to collect all the bounding boxes of the tree in a vector.

Parameters:

bBoxCollection Output vector that will be filled with the bounding boxes of the tree.

void AABBTree::collectNodes (std::vector< const AABBNode * > & *nodeCollection*) const

Class to collect all the nodes of the tree in a vector.

Parameters:

nodeCollection Output vector that will be filled with the nodes of the tree.

void AABBTree::collectOverlappingLeaves (const AABBTree * *otherTree*, std::vector< FacePair * > & *overlappingFacePairs*) const

Class that determines all the leaves, terminal nodes, of this tree that are in contact with another tree.

Parameters:

otherTree Other tree to detect overlaping leaves against.
overlappingFacePairs Output vector which will be filled with the face pairs that overlap.

void AABBTree::collectOverlappingLeaves (const AABBTree * *otherTree*, boost::ptr_vector< FacePair > & *overlappingFacePairs*) const

Class that determines all the leaves, terminal nodes, of this tree that are in contact with another tree.

Parameters:

otherTree Other tree to detect overlaping leaves against.
overlappingFacePairs Output vector which will be filled with the face pairs that overlap.

void AABBTree::collectSelfOverlappingLeaves (std::vector< FacePair * > & *overlappingFacePairs*) const

Class that determines all the leaves, terminal nodes, of this tree that overlap.

Parameters:

overlappingFacePairs Output vector which will be filled with the face pairs that overlap.

void AABBTree::collectSelfOverlappingLeaves (boost::ptr_vector< FacePair > & *overlappingFacePairs*) const

Class that determines all the leaves, terminal nodes, of this tree that overlap.

Parameters:

overlappingFacePairs Output vector which will be filled with the face pairs that overlap.

The documentation for this class was generated from the following files:

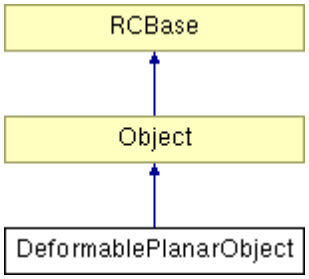
- src/PMDO/PMDO/AABB/AABBTree.h
- src/PMDO/PMDO/AABB/AABBTree.cpp

5.2.3 DeformablePlanarObject Class Reference

Class for a cloth object and all its atributes and methods.

`#include <DeformablePlanarObject.h>`

Inheritance diagram for DeformablePlanarObject:



Public Member Functions

DeformablePlanarObject (int fps, float gravity, RCShdPtr< GeMesh > initialMesh, const String &parFilename, const String &velConstraintFileName)

Constructor of the class. This loads the mesh, parameters and velocities of the object.

DeformablePlanarObject (const DeformablePlanarObject &dpo)

Copy constructor of the DPO.

DeformablePlanarObject & operator= (const DeformablePlanarObject &dpo)

Assignment operator.

virtual void step (BaTime::Duration At)

Function to step the simulation forward in time, NOT taking into account collisions.

RCShdPtr< SimSimulator > getSimulator ()

_stepper->_simulator accessor.

const RCShdPtr< SimSimulator > getSimulator () const

_stepper->_simulator accessor.

double getArea () const

_area accessor.

int getNFaces () const

_nFaces accessor.

virtual RCShdPtr< GeMesh > getMeshPtr ()

_simulator->_meshPtr accessor.

virtual const RCShdPtr< GeMesh > getMeshPtr () const

_simulator->_meshPtr constant accessor.

DpoVector getVelocity (const unsigned int vid) const

Interface to _simulator->getVelocity(vid) .

void setVelocity (unsigned int vid, const DpoVector &newVelocity)

Interface to _simulator->setVelocity(vid,newVelocity) .

virtual bool isVertexConstrained (unsigned int vertexId) const

---

Detailed Description

Class for a cloth object and all its atributes and methods.

Constructor & Destructor Documentation

DeformablePlanarObject::DeformablePlanarObject (int *fps*, float *gravity*, RCShdPtr< GeMesh > *initialMesh*, const String & *parFilename*, const String & *velConstraintFileName*)

Constructor of the class. This loads the mesh, parameters and velocities of the object.

Constructor of the class.

Parameters:

fps Frames per second of needed information about the mesh.
gravity Gravity.
initialMesh Initial mesh supplied to the simulator. TODO: At the moment no initial velocity can be supplied.
parFilename Parameters of the object (thickness, density and diffuse color)
velConstraintFileName Filename of the file that constains information about velocity constraints.

DeformablePlanarObject::DeformablePlanarObject (const DeformablePlanarObject & *dpo*)

Copy constructor of the DPO.

Member Function Documentation

DeformablePlanarObject & DeformablePlanarObject::operator= (const DeformablePlanarObject & *dpo*)

Assignment operator.

void DeformablePlanarObject::step (BaTime::Duration *At*) `[virtual]`

Function to step the simulation forward in time, NOT taking into account collisions.

The only stepper that takes into account collisions is *WorldStepStrategy* . This function takes care of velocity constraints loading before calling _stepper->step().

Parameters:

At Increment of time. Duration of time that must be simulated in ms.

Implements Object.

RCShdPtr< SimSimulator > DeformablePlanarObject::getSimulator ()

_stepper->_simulator accessor.

const RCShdPtr< SimSimulator > DeformablePlanarObject::getSimulator () const

_stepper->_simulator accessor.

double DeformablePlanarObject::getArea () const

_area accessor.

int DeformablePlanarObject::getNFaces () const

_nFaces accessor.

RCShdPtr< GeMesh > DeformablePlanarObject::getMeshPtr () `[inline, virtual]`

_simulator->_meshPtr accessor.

Implements Object.

const RCShdPtr< GeMesh > DeformablePlanarObject::getMeshPtr () const `[inline, virtual]`

_simulator->_meshPtr constant accessor.

Implements Object.

DpoVector DeformablePlanarObject::getVelocity (const unsigned int *vid*) const

Interface to *_simulator->getVelocity(vid)* .

void DeformablePlanarObject::setVelocity (unsigned int *vid*, const DpoVector & *newVelocity*)

Interface to *_simulator->setVelocity(vid,newVelocity)* .

bool DeformablePlanarObject::isVertexConstrained (unsigned int *vertexId*) const `[virtual]`

Implements Object.

The documentation for this class was generated from the following files:

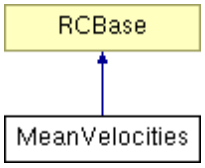
src/PMDO/PMDO/DeformablePlanarObject.h
src/PMDO/PMDO/DeformablePlanarObject.cpp

5.2.4 MeanVelocities Class Reference

Class containing information about the mean velocities of ONE object during a time-step.

`#include <MeanVelocities.h>`

Inheritance diagram for MeanVelocities:



Public Member Functions

MeanVelocities (RCShdPtr< GeMesh > initialMesh, RCShdPtr< GeMesh > finalMesh, float timestep)

Constructor of the class.

MeanVelocities (const MeanVelocities &rhs)

Copy constructor.

RCShdPtr< GeMesh > CreateFinalMesh ()

Method to create a final mesh given the initial state and the mean velocities.

void setVelocity (int id, GeVector &velocity)

Mutates the mean velocity of a vertex.

Public Attributes

std::vector< RCShdPtr< GeVector > > _meanVelocities

Mean velocities of the nodes during the time interval.

RCShdPtr< GeMesh > _initialMesh

Initial state of the mesh.

RCShdPtr< GeMesh > _finalMesh

Final state of the mesh.

float _timestep

Time interval.

Detailed Description

Class containing information about the mean velocities of ONE object during a time-step.

This class holds information about the mean velocity of the nodes of a mesh given an initial state and a final state in a time interval. The impulses for collision and contact response affect the velocities contained in this class.

Constructor & Destructor Documentation

MeanVelocities::MeanVelocities (RCShdPtr< GeMesh > *initialMesh*, RCShdPtr< GeMesh > *finalMesh*, float *timestep*)

Constructor of the class.

Given the input parameters, the constructor calculates the mean velocities during a time-step and saves them.

Parameters:

initialMesh Initial state of the mesh (at the beginning of the time interval).
finalMesh Final state of the mesh (at the end of the interval).
timestep Time interval (in seconds).

MeanVelocities::MeanVelocities (const MeanVelocities & *rhs*)

Copy constructor.

Member Function Documentation

RCShdPtr< GeMesh > MeanVelocities::CreateFinalMesh ()

Method to create a final mesh given the initial state and the mean velocities.

void MeanVelocities::setVelocity (int *id*, GeVector & *velocity*)

Mutates the mean velocity of a vertex.

Parameters:

id Id of the vertex whose velocity is changed.
velocity New velocity of the verex.

Member Data Documentation

std::vector<RCShdPtr<GeVector > > MeanVelocities::_meanVelocities

Mean velocities of the nodes during the time interval.

RCShdPtr< GeMesh > MeanVelocities::_initialMesh

Initial state of the mesh.

RCShdPtr< GeMesh > MeanVelocities::_finalMesh

Final state of the mesh.

float MeanVelocities::_timestep

Time interval.

The documentation for this class was generated from the following files:

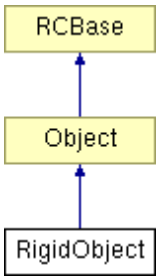
src/PMDO/PMDO/MeanVelocities.h
src/PMDO/PMDO/MeanVelocities.cpp

5.2.5 RigidObject Class Reference

Class for a rigid object and all its atributes and methods.

`#include <RigidObject.h>`

Inheritance diagram for RigidObject:



Public Member Functions

RigidObject (const String &meshFilename, const String &parFilename, const String &velConstraintsFilename)
Constructor of the class. This loads the mesh, parameters and velocities of the object.

RigidObject (const RigidObject &rhs)
Copy constructor.

const RigidObject & operator= (const RigidObject &rhs)
Assignment operator.

void createSoGroup (const SbColor &color)
Function that creates a SoGroup of the mesh to display on a coin widget.

virtual RCShdPtr< GeMesh > getMeshPtr ()
Accessor to the mesh of the object.

virtual const RCShdPtr< GeMesh > getMeshPtr () const
Accessor to the mesh of the object.

virtual void step (BaTime::Duration At)
Method to simulate the rigid object forward in time. (Doesn't take into account collisions).

boost::shared_ptr< std::vector< RCShdPtr< FaceId > > > calcFacesList ()
Method that returns a list with all the faces of the rigid object.

void move (const GeVector &translation)
Move the object.

virtual bool isVertexConstrained (unsigned int vertexId) const
Returns whether a vertex's velocity is constrained for the current step.

Detailed Description

Class for a rigid object and all its atributes and methods.

Constructor & Destructor Documentation

RigidObject::RigidObject (const String & *meshFilename*, const String & *parFilename*, const String & *velConstraintsFilename*)

Constructor of the class. This loads the mesh, parameters and velocities of the object.

Constructor of the class.

Parameters:

meshFilename Initial mesh supplied to the simulator. TODO: At the moment no initial velocity can be supplied.
parFilename Parameters of the object (thickness, mass and diffuse color) Thickness and mass should be zero.
velConstraintFileName Filename of the file that constains information about velocity constraints.

RigidObject::RigidObject (const RigidObject & *rhs*)

Copy constructor.

Member Function Documentation

const RigidObject & RigidObject::operator= (const RigidObject & *rhs*)

Assignment operator.

void RigidObject::createSoGroup (const SbColor & *color*)

Function that creates a SoGroup of the mesh to display on a coin widget.

Parameters:

color Diffuse color of the displayed mesh.

Define the normals used:

Define coordinates for vertices

Define the FaceSet

RCShdPtr< GeMesh > RigidObject::getMeshPtr () `[virtual]`

Accessor to the mesh of the object.

Implements Object.

const RCShdPtr< GeMesh > RigidObject::getMeshPtr () const `[virtual]`

Accessor to the mesh of the object.

Implements Object.

void RigidObject::step (BaTime::Duration *At*) `[virtual]`

Method to simulate the rigid object forward in time. (Doesn't take into account collisions).

Implements Object.

boost::shared_ptr< std::vector< RCShdPtr< FaceId > > > RigidObject::calcFacesList ()

Method that returns a list with all the faces of the rigid object.

void RigidObject::move (const GeVector & *translation*)

Move the object.

Parameters:

translation Amount to move in x, y and z direction.

bool RigidObject::isVertexConstrained (unsigned int *vertexId*) const `[virtual]`

Returns whether a vertex's velocity is constrained for the current step.

Parameters:

vertexId Id of the vertex in the mesh of the rigid object

Implements Object.

The documentation for this class was generated from the following files:

src/PMDO/PMDO/RigidObject.h
src/PMDO/PMDO/RigidObject.cpp

5.2.6 StlMeshLoader Class Reference

Class that loads a mesh from a STL file and creates a freecloth::GeMesh.

`#include <StlMeshLoader.h>`

Public Member Functions

StlMeshLoader (String filename)

Constructor of the class.

RCShdPtr< GeMesh > getMeshPtr ()

Accessor to the loaded mesh.

Detailed Description

Class that loads a mesh from a STL file and creates a freecloth::GeMesh.

Constructor & Destructor Documentation

StlMeshLoader::StlMeshLoader (String *filename*)

Constructor of the class.

The constructor load the mesh from a STL file and saves it for later access in *_mesh* .

Parameters:

filename Name of the STL file, taking into account the current directory.

Member Function Documentation

RCShdPtr< GeMesh > StlMeshLoader::getMeshPtr ()

Accessor to the loaded mesh.

The documentation for this class was generated from the following files:

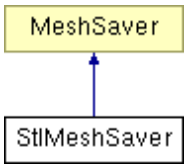
src/PMDO/PMDO/StlMeshLoader.h
src/PMDO/PMDO/StlMeshLoader.cpp

5.2.7 StlMeshSaver Class Reference

Class that saves a mesh in STL format.

`#include <StlMeshSaver.h>`

Inheritance diagram for StlMeshSaver:



Public Member Functions

StlMeshSaver (GeMesh &mesh, String meshName)

Constructor of the class.

virtual void save (String fileName)

Method to save the mesh given to the constructor.

Detailed Description

Class that saves a mesh in STL format.

Constructor & Destructor Documentation

StlMeshSaver::StlMeshSaver (GeMesh & *mesh*, String *meshName*)

Constructor of the class.

The constructor contains a mesh for later saving.

Parameters:

mesh Mesh to be saved later.
meshName Name of the mesh that will appear in the STL file.

Member Function Documentation

void StlMeshSaver::save (String *fileName*) `[virtual]`

Method to save the mesh given to the constructor.

Parameters:

fileName Name of the STL file that will be created, taking into account the current directory

Implements MeshSaver.

The documentation for this class was generated from the following files:

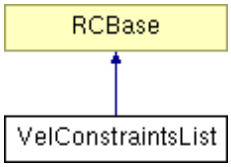
src/PMDO/PMDO/StlMeshSaver.h
src/PMDO/PMDO/StlMeshSaver.cpp

5.2.8 VelConstraintsList Class Reference

This class contains a list of velocity constraints, to be applied to an object, read from a file.

`#include <VelConstraints.h>`

Inheritance diagram for VelConstraintsList:



Public Member Functions

VelConstraintsList (String filename)

Constructor.

const VelConstraints getVelConstraints (float time) const

Method to retrieve VelocityConstraints corresponding to a certain time.

String getFilename ()

_filename accessor.

Detailed Description

This class contains a list of velocity constraints, to be applied to an object, read from a file.

Constructor & Destructor Documentation

VelConstraintsList::VelConstraintsList (String *filename*)

Constructor.

This constructor already reads and stores data from a file with the format: (example)

Time 0 Number of constraint vectors 1 VertexId 0 Vector [0 0 0]

Time 1 Number of constraint vectors 1 VertexId 0 Vector [0 0 0.1]

Time 2 Number of constraint vectors 0

Time 2.5 Number of constraint vectors 2 VertexId 15 Vector [0.1 -0.1 0] VertexId 0 Vector [0.1 -0.1 0]

Explanation of the example: From time 0 to 1 the vertex 0 will be fixed. From time 1 to 2 the vertex 0 will have a velocity of [0 0 0.1] From time 2 to 2.5 no vertexs will be constraint. From time 2.5 on vertexs 15 and 0 will be constraint with a velocity of [0.1 -0.1 0].

NOTE: Only numbers are read, but they must have coeherence or the simulation will fail.

Parameters:

filename File containing information to be stored.

Member Function Documentation

const VelConstraints VelConstraintsList::getVelConstraints (float *time*) const

Method to retrieve VelocityConstraints corresponding to a certain time.

String VelConstraintsList::getFilename ()

_filename accessor.

The documentation for this class was generated from the following files:

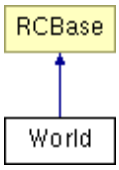
src/PMDO/PMDO/VelConstraints.h
src/PMDO/PMDO/VelConstraints.cpp

5.2.9 World Class Reference

Class that contains all the objects in the scene and can simulate their behaviour.

`#include <World.h>`

Inheritance diagram for World:



Public Member Functions

World ()
Constructor.

World (const World &world)
Copy constructor of the world.

World & operator= (const World &world)
Assignment operator of the world.

~World ()
Destructor.

void stepAllowingCollisions ()
Steps the simulation forward without taking into account contacts or collisions.

void step ()
Steps the simulation forward taking into account contacts or collisions.

void rewind ()

Restarts the world loading again from the files.

BaTime::Instant getTime ()

Simulation time accessor.

void setTime (BaTime::Instant time)

Simulation time mutator.

int getFps ()

Fps of the simulation accessor.

RCShdPtr< WorldObjects > getObjectsPtr ()

Accessor to all the objects in the world.

void setObjectsPtr (const RCShdPtr< WorldObjects > objects)

Accessor to all the objects in the world.

void setSaveIterationMeshes (bool saveIterationMeshes)

Method that allows the caller to save the iterationMeshes or not.

Friends

class WorldObjects

Classes

class Parameters

Class that contains the parameter of the world loaded from a file.

class WorldStepStrategy

A stepper that takes into account collisions.

Detailed Description

Class that contains all the objects in the scene and can simulate their behaviour.

This class is the one that a developer can most easily use. It can load a scene, contain objects, velocity restrictions and parameters, and it can also simulate the scene.

Constructor & Destructor Documentation

World::World ()

Constructor.

The constructor reads the information stored in the default path and loads parameters and objects from it.

World::World (const World & *world*)

Copy constructor of the world.

World::~World ()

Destructor.

Member Function Documentation

World& World::operator= (const World & *world*)

Assignment operator of the world.

void World::stepAllowingCollisions (void)

Steps the simulation forward without taking into account contacts or collisions.

void World::step (void)

Steps the simulation forward taking into account contacts or collisions.

void World::rewind ()

Restarts the world loading again from the files.

BaTime::Instant World::getTime () `[inline]`

Simulation time accessor.

void World::setTime (BaTime::Instant *time*)

Simulation time mutator.

int World::getFps () `[inline]`

Fps of the simulation accessor.

RCShdPtr< WorldObjects > World::getObjectsPtr () `[inline]`

Accessor to all the objects in the world.

void World::setObjectsPtr (const RCShdPtr< WorldObjects > *objects*)

Accessor to all the objects in the world.

(Used to asign new objects to the world)

void World::setSaveIterationMeshes (bool *saveIterationMeshes*)

Method that allows the caller to save the iterationMeshes or not.

Friends And Related Function Documentation

friend class WorldObjects `[friend]`

The documentation for this class was generated from the following files:

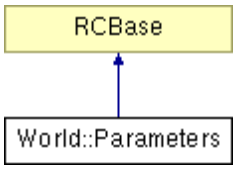
src/PMDO/PMDO/World.h
src/PMDO/PMDO/World.cpp

5.2.10 World::Parameters Class Reference

Class that contains the parameter of the world loaded from a file.

`#include <World.h>`

Inheritance diagram for World::Parameters:



Public Member Functions

Parameters ()
Constructor. The constructor initialises the parameters with zeros.

void loadFromFile ()
Method that loads parameters from a file and stores them.

Public Attributes

float _gravity
Parameter: Gravity.

int _fps
Parameter: Frames per second requiered.

Detailed Description

Class that contains the parameter of the world loaded from a file.

Constructor & Destructor Documentation

World::Parameters::Parameters ()

Constructor. The constructor initialises the parameters with zeros.

Member Function Documentation

void World::Parameters::loadFromFile ()

Method that loads parameters from a file and stores them.

Member Data Documentation

float World::Parameters::_gravity

Parameter: Gravity.

int World::Parameters::_fps

Parameter: Frames per second requiered.

The documentation for this class was generated from the following files:

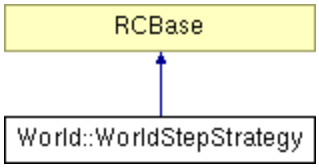
src/PMDO/PMDO/World.h
src/PMDO/PMDO/World.cpp

5.2.11 World::WorldStepStrategy Class Reference

A stepper that takes into account collisions.

`#include <World.h>`

Inheritance diagram for World::WorldStepStrategy:



Public Member Functions

WorldStepStrategy (World *world)

Contructor.

void stepAllowingCollisions ()

Step that doesn't take into account collisions.

void step ()

Step that takes into account collisions.

BaTime::Instant getTime () const

Accesor to the time of the simulaton.

Detailed Description

A stepper that takes into account collisions.

Class similar to SimStepStrategyAdaptive but focusing on collisions. The stepping strategy could be divided in three layers (each with its own nomenclature):

-->A) Step layer:

This is the top layer. A *step* is a user step, which means that the timestep is 1.0/frameRate. This layer calls the colStep layer for smaller steps which, in turn, calls the bottom subStep layer. A step >= colStep >= subStep.

-->B) colStep layer:

This is the middle layer, called by the Step layer once or more times per frame and calls subStep layer once or more times per colStep. This is a collision step, which means that after every colStep the path is checked for collisions and, if there have been collisions, must either be halved or process collision impulses. As explained in 5 [Brid'02].

-->C) subStep layer:

This is the bottom layer. SubSteps call internal simulation (not taking into account collisions) substeps. The subStep layer is also adaptive (inherited from SimStepStrategy).

Constructor & Destructor Documentation

World::WorldStepStrategy::WorldStepStrategy (World * *world*)

Contructor.

Parameters:

world World to be simulated.

Member Function Documentation

void World::WorldStepStrategy::stepAllowingCollisions ()

Step that doesn't take into account collisions.

Parameters:

At Duration of time to be simulated.

void World::WorldStepStrategy::step ()

Step that takes into account collisions.

BaTime::Instant World::WorldStepStrategy::getTime () const

Accesor to the time of the simulaton.

The documentation for this class was generated from the following files:

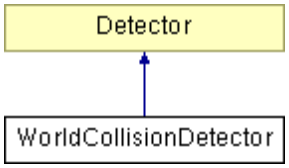
src/PMDO/PMDO/World.h
src/PMDO/PMDO/WorldStepStrategy.cpp

5.2.12 WorldCollisionDetector Class Reference

Class that detects the collisions that take place in the scene during a simulated step.

`#include <WorldCollisionDetector.h>`

Inheritance diagram for WorldCollisionDetector:



Public Member Functions

WorldCollisionDetector (const RCShdPtr< WorldObjects > initialWorldObjects, const RCShdPtr< WorldMeanVelocities > worldMeanVelocities)

Constructor of the class.

WorldCollisionDetector (const RCShdPtr< WorldObjects > initialWorldObjects, const RCShdPtr< WorldObjects > finalWorldObjects, float timestep)

Constructor of the class.

virtual const DetectionPair * calcFirstToSolve () const

Finds the first DetectionPair that should be solved (in collision it is the first to collide).

Detailed Description

Class that detects the collisions that take place in the scene during a simulated step.

When constructed, this class atomatically detects collision pairs, which can be retreived later with the method *Detector::getProposedPairs()* .

Constructor & Destructor Documentation

WorldCollisionDetector::WorldCollisionDetector (const RCShdPtr< WorldObjects > *initialWorldObjects*, const RCShdPtr< WorldMeanVelocities > *worldMeanVelocities*)

Constructor of the class.

This constructor detects collisions between the objects in the scene between an initial and a proposed final position.

Parameters:

initialWorldObjects Initial state, positions, of every object in the scene.
worldMeanVelocities Proposed mean velocities of every object in the scene.

WorldCollisionDetector::WorldCollisionDetector (const RCShdPtr< WorldObjects > *initialWorldObjects*, const RCShdPtr< WorldObjects > *finalWorldObjects*, float *timestep*)

Constructor of the class.

This constructor detects collisions between the objects in the scene between an initial and a proposed final position.

Parameters:

initialWorldObjects Initial state, positions, of every object in the scene.
finalWorldObjects Proposed final state, positions, of every object in the scene.
timestep Duration of the step taken.

Member Function Documentation

const DetectionPair * WorldCollisionDetector::calcFirstToSolve () const `[virtual]`

Finds the first DetectionPair that should be solved (in collision it is the first to collide).

If there are no pairs it returns 0. Used in serial update.

Implements Detector.

The documentation for this class was generated from the following files:

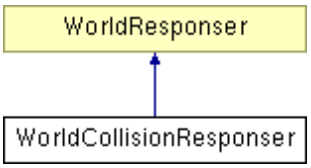
src/PMDO/PMDO/Detection/WorldCollisionDetector.h
src/PMDO/PMDO/Detection/WorldCollisionDetector.cpp

5.2.13 WorldCollisionResponser Class Reference

Class that automatically applies all the collision impulses required in a scene.

```
#include <WorldCollisionResponser.h>
```

Inheritance diagram for WorldCollisionResponser:



Public Member Functions

WorldCollisionResponser (RCShdPtr< WorldMeanVelocities > worldMeanVelocities, const WorldCollisionDetector &worldCollisionDetector, const PositionUpdater positionUpdater, const VelocityUpdater velocityUpdater)

Constructor of the class.

Protected Member Functions

virtual void addImpulses (const Detector &detector)

Function that adds impulses to the mean velocities of the objects according to detected collisions.

virtual void addPTImpulse (const PTPair &ptPair)

Function that adds impulses to a point-triangle pair which are going to collide.

virtual void addEEImpulse (const EEPair &eePair)

Function that adds impulses to a edge-edge pair which are going to collide.

Detailed Description

Class that automatically applies all the collision impulses required in a scene.

This class works pretty much like a function, as the constructor does everything and no other method should be called.

Constructor & Destructor Documentation

WorldCollisionResponser::WorldCollisionResponser (RCShdPtr< WorldMeanVelocities > *worldMeanVelocities*, const WorldCollisionDetector & *worldCollisionDetector*, const PositionUpdater *positionUpdater*, const VelocityUpdater *velocityUpdater*)

Constructor of the class.

This constructor applies collision impulses to *worldMeanVelocities* according to the collisions detected in *worldCollisionDetector* .

Parameters:

worldMeanVelocities Proposed mean velocities of every object in the scene. These will be modified.
worldCollisionDetector *WorldCollisionDetector* object with information about detected collisions.
positionUpdater Method of applying impulses. PARALLEL update (recommended, faster), which makes an average of the impulses applied to every node or SERIAL update, much slower.
velocityUpdater Method of updating velocities. IMPLICIT update (recommended, stable), which updates resolving an implicit equation or CONST_ACCELERATION update (unstable), which rsolves explicitly supossing a constant acceleration during the step.

Member Function Documentation

void WorldCollisionResponser::addImpulses (const Detector & *detector*) [protected, virtual]

Function that adds impulses to the mean velocities of the objects according to detected collisions.

Parameters:

detector Object with detected collisions.

Implements WorldResponser.

void WorldCollisionResponser::addPTImpulse (const PTPair & *ptPair*) [protected, virtual]

Function that adds impulses to a point-triangle pair which are going to collide.

Parameters:

ptPair Point-triangle pair.

Implements WorldResponser.

void WorldCollisionResponser::addEEImpulse (const EEPair & *eePair*) [protected, virtual]

Function that adds impulses to a edge-edge pair which are going to collide.

Parameters:

eePair Edge-edge pair.

Implements WorldResponser.

The documentation for this class was generated from the following files:

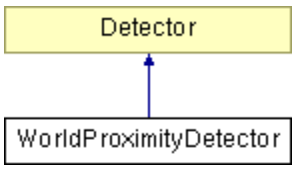
src/PMDO/PMDO/Responser/WorldCollisionResponser.h
src/PMDO/PMDO/Responser/WorldCollisionResponser.cpp

5.2.14 WorldProximityDetector Class Reference

Class that detects the contacts that take place in the scene during a simulated step.

`#include <WorldProximityDetector.h>`

Inheritance diagram for WorldProximityDetector:



Public Member Functions

WorldProximityDetector (const RCShdPtr< WorldObjects > worldObjects, const RCShdPtr< WorldMeanVelocities > worldMeanVelocities)

Constructor of the class.

virtual const DetectionPair * calcFirstToSolve () const

Finds the first DetectionPair that should be solved (in contact it is the nearest).

Detailed Description

Class that detects the contacts that take place in the scene during a simulated step.

When constructed, this class atomatically detects contact pairs, which can be retreived later with the method *Detector::getProposedPairs()* .

Constructor & Destructor Documentation

WorldProximityDetector::WorldProximityDetector (const RCShdPtr< WorldObjects > *worldObjects*, const RCShdPtr< WorldMeanVelocities > *worldMeanVelocities*)

Constructor of the class.

This constructor detects contacts between the objects in the scene from an initial position.

Parameters:

worldObjects Initial state, positions, of every object in the scene.
worldMeanVelocities Proposed mean velocities of every object in the scene.

Member Function Documentation

const DetectionPair * WorldProximityDetector::calcFirstToSolve () const [virtual]

Finds the first DetectionPair that should be solved (in contact it is the nearest).

If there are no pairs it returns 0. Used in serial update.

Implements Detector.

The documentation for this class was generated from the following files:

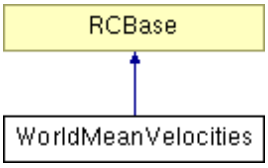
src/PMDO/PMDO/Detection/WorldProximityDetector.h
src/PMDO/PMDO/Detection/WorldProximityDetector.cpp

5.2.15 WorldMeanVelocities Class Reference

This class calculates and holds the mean velocities of every object in the scene during a simulated step.

`#include <WorldMeanVelocities.h>`

Inheritance diagram for WorldMeanVelocities:



Public Member Functions

WorldMeanVelocities (const RCShdPtr< WorldObjects > initialObjects, const RCShdPtr< WorldObjects > finalObjects, float timestep)

Constructor of the class.

WorldMeanVelocities (const WorldMeanVelocities &rhs)

Copy constructor of the class.

const RCShdPtr< MeanVelocities > getMeanVelocities (const Object *object) const

Method that returns the mean velocities of an object. The object ptr is used as a keyword to map its mean velocities.

RCShdPtr< MeanVelocities > getMeanVelocities (const Object *object)

Method that returns the mean velocities of an object. The object ptr is used as a keyword to map its mean velocities.

void applyMeanVelocities (RCShdPtr< WorldObjects > objects, const float timestep) const

Method that calculates the final positions of every object in the scene according to the mean velocities.

std::map< const Object *, std::vector< GeVector > > calcZeroVectorMap () const

Creates a map with zero velocities.

std::map< const Object *, std::vector< unsigned int > > calcZeroIntMap ()

Creates a map from (object0, 0), (object1, 0),..., (object(n-1), 0).

float getTimestep () const

Accessor to the duration of the simulated step.

const RCShdPtr< WorldObjects > calcFinalPositions (const RCShdPtr< WorldObjects > initialWorldObjects) const

Calculates final positions and returns a pointer the world objects. This doesn't modify the final velocities.

Detailed Description

This class calculates and holds the mean velocities of every object in the scene during a simulated step.

When constructed, this class atomatically calculates the mean velocities, which can be later retreived with *getMeanVelocities()* or applied to calculate the final positions with *calcFinalPositions* .

Constructor & Destructor Documentation

WorldMeanVelocities::WorldMeanVelocities (const RCShdPtr< WorldObjects > *initialObjects*, const RCShdPtr< WorldObjects > *finalObjects*, float *timestep*)

Constructor of the class.

This constructor calculates the mean velocities of the objects: (final positions - initial positions)/timestep.

Parameters:

initialObjects Initial state, positions, of every object in the scene.
finalObjects Proposed final state, positions, of every object in the scene.
timestep Duration of the step taken.

WorldMeanVelocities::WorldMeanVelocities (const WorldMeanVelocities & *rhs*)

Copy constructor of the class.

Member Function Documentation

const RCShdPtr< MeanVelocities > WorldMeanVelocities::getMeanVelocities (const Object * *object*) const

Method that returns the mean velocities of an object. The object ptr is used as a keyword to map its mean velocities.

Parameters:

object Object whose mean velocities must be returned.

Returns:

Object's mean velocities.

RCShdPtr< MeanVelocities > WorldMeanVelocities::getMeanVelocities (const Object * *object*)

Method that returns the mean velocities of an object. The object ptr is used as a keyword to map its mean velocities.

Parameters:

object Object whose mean velocities must be returned.

Returns:

Object's mean velocities.

void WorldMeanVelocities::applyMeanVelocities (RCShdPtr< WorldObjects > *objects*, const float *timestep*) const

Method that calculates the final positions of every object in the scene according to the mean velocities.

Objects are substituted with the final objects.

Parameters:

object Object whose mean velocities must be returned.
timestep Duration of the step taken.

std::map< const Object *, std::vector< GeVector > > WorldMeanVelocities::calcZeroVectorMap () const

Creates a map with zero velocities.

Internally used by the program.

Returns:

A copy of the mean velocities, but with every velocity nullified.

std::map< const Object *, std::vector< unsigned int > > WorldMeanVelocities::calcZeroIntMap ()

Creates a map from (object0, 0), (object1, 0),..., (object(n-1), 0).

Used internally by the program.

float WorldMeanVelocities::getTimestep () const

Accessor to the duration of the simulated step.

const RCShdPtr< WorldObjects > WorldMeanVelocities::calcFinalPositions (const RCShdPtr< WorldObjects > *initialWorldObjects*) const

Calculates final positions and returns a pointer the world objects. This doesn't modify the final velocities.

Function similar to *applyMeanVelocities* .

Parameters:

initialWorldObjects State of the objects in the scene at the beginning of the step.

Returns:

Final state of the object in the scene according to the mean velocities stored in the class.

The documentation for this class was generated from the following files:

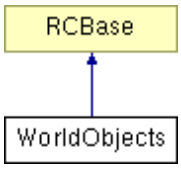
src/PMDO/PMDO/WorldMeanVelocities.h
src/PMDO/PMDO/WorldMeanVelocities.cpp

5.2.16 WorldObjects Class Reference

Class that manages all the objects in world.

`#include <WorldObjects.h>`

Inheritance diagram for WorldObjects:



Public Member Functions

WorldObjects (World *world)

Constructor that initializes with empty vectors.

WorldObjects (const WorldObjects &rhs)

Copy constructor.

const WorldObjects & operator= (const WorldObjects &rhs)

Assignment operator.

~WorldObjects ()

Destructor.

void loadFromFiles ()

Method that loads objects from files.

std::vector< RCShdPtr< DeformablePlanarObject > > & getDpos ()

_dpos accessor.

const std::vector< RCShdPtr< DeformablePlanarObject > > & getDpos () const

_dpos constant accessor.

std::vector< RCShdPtr< RigidObject > > & getRos ()

_ros accessor.

const std::vector< RCShdPtr< RigidObject > > & getRos () const
_ros constant accessor.

World * getWorld ()
Accessor to the parent world.

const World *const getWorld () const
Accessor to the parent world.

void stepAllowingCollisions (BaTime::Duration At)
Simulate the objects one step forward without taking into account contacts or collisions.

Detailed Description

Class that manages all the objects in world.

Constructor & Destructor Documentation

WorldObjects::WorldObjects (World * *world*)

Constructor that initializes with empty vectors.

Parameters:

world World that contains the objects in WorldObjects.

WorldObjects::WorldObjects (const WorldObjects & *rhs*)

Copy constructor.

WorldObjects::~WorldObjects ()

Destructor.

Member Function Documentation

const WorldObjects & WorldObjects::operator= (const WorldObjects & *rhs*)

Assignment operator.

void WorldObjects::loadFromFiles ()

Method that loads objects from files.

std::vector< RCShdPtr< DeformablePlanarObject > > & WorldObjects::getDpos ()

_dpos accessor.

const std::vector< RCShdPtr< DeformablePlanarObject > > & WorldObjects::getDpos () const

_dpos constant accessor.

std::vector< RCShdPtr< RigidObject > > & WorldObjects::getRos ()

_ros accessor.

const std::vector< RCShdPtr< RigidObject > > & WorldObjects::getRos () const

_ros constant accessor.

World* WorldObjects::getWorld ()

Accessor to the parent world.

const World* const WorldObjects::getWorld () const

Accessor to the parent world.

void WorldObjects::stepAllowingCollisions (BaTime::Duration *At*)

Simulate the objects one step forward without taking into account contacts or collisions.

The documentation for this class was generated from the following files:

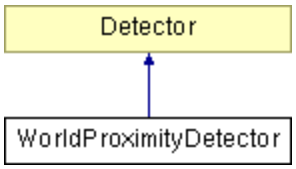
src/PMDO/PMDO/WorldObjects.h
src/PMDO/PMDO/World.cpp
src/PMDO/PMDO/WorldObjects.cpp

5.2.17 WorldProximityDetector Class Reference

Class that detects the contacts that take place in the scene during a simulated step.

`#include <WorldProximityDetector.h>`

Inheritance diagram for WorldProximityDetector:



Public Member Functions

WorldProximityDetector (const RCShdPtr< WorldObjects > worldObjects, const RCShdPtr< WorldMeanVelocities > worldMeanVelocities)

Constructor of the class.

virtual const DetectionPair * calcFirstToSolve () const

Finds the first DetectionPair that should be solved (in contact it is the nearest).

Detailed Description

Class that detects the contacts that take place in the scene during a simulated step.

When constructed, this class atomatically detects contact pairs, which can be retreived later with the method *Detector::getProposedPairs()* .

Constructor & Destructor Documentation

WorldProximityDetector::WorldProximityDetector (const RCShdPtr< WorldObjects > *worldObjects*, const RCShdPtr< WorldMeanVelocities > *worldMeanVelocities*)

Constructor of the class.

This constructor detects contacts between the objects in the scene from an initial position.

Parameters:

worldObjects Initial state, positions, of every object in the scene.
worldMeanVelocities Proposed mean velocities of every object in the scene.

Member Function Documentation

const DetectionPair * WorldProximityDetector::calcFirstToSolve () const `[virtual]`

Finds the first DetectionPair that should be solved (in contact it is the nearest).

If there are no pairs it returns 0. Used in serial update.

Implements Detector.

The documentation for this class was generated from the following files:

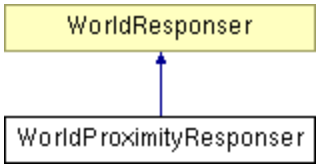
src/PMDO/PMDO/Detection/WorldProximityDetector.h
src/PMDO/PMDO/Detection/WorldProximityDetector.cpp

5.2.18 WorldProximityResponser Class Reference

Class that automatically applies all the contact impulses required in a scene.

`#include <WorldProximityResponser.h>`

Inheritance diagram for WorldProximityResponser:



Public Member Functions

WorldProximityResponser (const RCShdPtr< WorldMeanVelocities > worldMeanVelocities, const WorldProximityDetector &worldProximityDetector, PositionUpdater positionUpdater, VelocityUpdater velocityUpdater)

Constructor of the class.

Protected Member Functions

virtual void addImpulses (const Detector &detector)

Function that adds impulses to the mean velocities of the objects according to detected contacts.

virtual void addPTImpulse (const PTPair &ptPair)

Function that adds impulses to a point-triangle pair which are in contact.

virtual void addEEImpulse (const EEPair &eePair)

Function that adds impulses to a edge-edge pair which are in contact.

Detailed Description

Class that automatically applies all the contact impulses required in a scene.

This class works pretty much like a function, as the constructor does everything and no other method should be called.

Constructor & Destructor Documentation

WorldProximityResponser::WorldProximityResponser (const RCShdPtr< WorldMeanVelocities > *worldMeanVelocities*, const WorldProximityDetector & *worldProximityDetector*, PositionUpdater *positionUpdater*, VelocityUpdater *velocityUpdater*)

Constructor of the class.

This constructor applies contac impulses to *worldMeanVelocities* according to the contacts detected in *worldContactDetector* .

Parameters:

worldMeanVelocities Proposed mean velocities of every object in the scene. These will be modified.
worldProximityDetector WorldProximityDetector object with information about detected contacts.
positionUpdater Method of applying impulses. PARALLEL update (recommended, faster), which makes an average of the impulses applied to every node or SERIAL update, much slower.
velocityUpdater Method of updating velocities. IMPLICIT update (recommended, stable), which updates resolving an implicit equation or CONST_ACCELERATION update (unstable), which rsolves explicitly supossing a constant acceleration during the step.

Member Function Documentation

void WorldProximityResponser::addImpulses (const Detector & *detector*) [protected, virtual]

Function that adds impulses to the mean velocities of the objects according to detected contacts.

Parameters:

detector Object with detected contacts.

Implements WorldResponser.

void WorldProximityResponser::addPTImpulse (const PTPair & *ptPair*) [protected, virtual]

Function that adds impulses to a point-triangle pair which are in contact.

Parameters:

ptPair Point-triangle pair.

Implements WorldResponser.

void WorldProximityResponser::addEEImpulse (const EEPair & *eePair*) [protected, virtual]

Function that adds impulses to a edge-edge pair which are in contact.

Parameters:

eePair Edge-edge pair.

Implements WorldResponser.

The documentation for this class was generated from the following files:

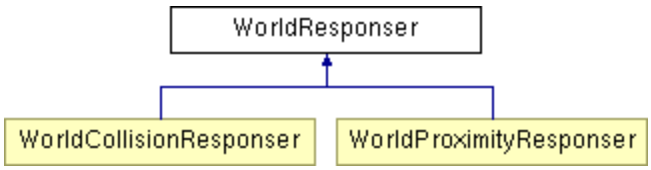
src/PMDO/PMDO/Responser/WorldProximityResponser.h
src/PMDO/PMDO/Responser/WorldProximityResponser.cpp

5.2.19 WorldResponser Class Reference

Class that contains functions that *WorldCollisionResponser* and *WorldProximityResponser* have in common.

`#include <WorldResponser.h>`

Inheritance diagram for WorldResponser:



Public Member Functions

WorldResponser (RCShdPtr< WorldMeanVelocities > worldMeanVelocities, const PositionUpdater positionUpdater, const VelocityUpdater velocityUpdater)

Constructor of the class.

void applyMutatedStepTo (const WorldObjects &initialObjects, WorldObjects &proposedObjects) const

Function that applies collision or contact impulses to initialObjects to obtain the final objects.

RCShdPtr< WorldMeanVelocities > getWorldMeanVelocities () const

Accessor to the mean velocities of every object in the world.

Protected Member Functions

virtual void addImpulses (const Detector &detector)=0

Pure virtual function that adds impulses to the mean velocities of the objects according to detected contacts or collisions.

virtual void addPTImpulse (const PTPair &ptPair)=0

Pure virtual function that adds impulses to a point-triangle pair which are in contact or going to collide.

virtual void addEEImpulse (const EEPair &eePair)=0

Pure virtual function that adds impulses to a edge-edge pair which are in contact or going to collide.

void distributePTImpulse (const float Ibar, const PTPair &ptPair)

Function that distributes an impulse, applied to a point-triangle pair, to the nodes.

void distributeEEImpulse (const float Ibar, const EEPair &eePair)

Function that distributes an impulse, applied to an edge-edge pair, to the nodes.

PositionUpdater getPositionUpdater () const

_positionUpdater object accessor.

VelocityUpdater getVelocityUpdater () const

_velocityUpdater object accessor.

VelocitiesMutator & getVelocitiesMutator ()

_velocityMutator object accessor.

Classes

class VelocitiesMutator

Nested internally used to change the mean velocity of the nodes in the scene.

Detailed Description

Class that contains functions that *WorldCollisionResponser* and *WorldProximityResponser* have in common.

Constructor & Destructor Documentation

WorldResponser::WorldResponser (RCShdPtr< WorldMeanVelocities > *worldMeanVelocities*, const PositionUpdater *positionUpdater*, const VelocityUpdater *velocityUpdater*)

Constructor of the class.

Parameters:

worldMeanVelocities Proposed mean velocities of every object in the scene. These will be modified.
positionUpdater Method of applying impulses. PARALLEL update (recommended, faster), which makes an average of the impulses applied to every node or SERIAL update, much slower.
velocityUpdater Method of updating velocities. IMPLICIT update (recommended, stable), which updates resolving an implicit equation or CONST_ACCELERATION update (unstable), which rsolves explicitly supossing a constant acceleration during the step.

Member Function Documentation

void WorldResponser::applyMutatedStepTo (const WorldObjects & *initialObjects*, WorldObjects & *proposedObjects*) const

Function that applies collision or contact impulses to *initialObjects* to obtain the final objects.

Parameters:

initialObjects Initial position of the objects in the scene.
proposedObjects Proposed final positions of the objects in the scene. These will be replaced with the actual final positions.

RCShdPtr< WorldMeanVelocities > WorldResponser::getWorldMeanVelocities () const

Accessor to the mean velocities of every object in the world.

Returns:

Mean velocities the objects in the scene.

virtual void WorldResponser::addImpulses (const Detector & *detector*) [protected, pure virtual]

Pure virtual function that adds impulses to the mean velocities of the objects according to detected contacts or collisions.

Parameters:

detector Object with detected collisions.

Implemented in WorldCollisionResponser, and WorldProximityResponser.

virtual void WorldResponser::addPTImpulse (const PTPair & *ptPair*) [protected, pure virtual]

Pure virtual function that adds impulses to a point-triangle pair which are in contact or going to collide.

Parameters:

ptPair Point-triangle pair.

Implemented in WorldCollisionResponser, and WorldProximityResponser.

virtual void WorldResponser::addEEImpulse (const EEPair & *eePair*) [protected, pure virtual]

Pure virtual function that adds impulses to a edge-edge pair which are in contact or going to collide.

Parameters:

eePair Edge-edge pair.

Implemented in WorldCollisionResponser, and WorldProximityResponser.

void WorldResponser::distributePTImpulse (const float *Ibar*, const PTPair & *ptPair*) [protected]

Function that distributes an impulse, applied to a point-triangle pair, to the nodes.

Parameters:

Ibar Total impulse to be applied to the point and to a specific point of the triangle. This impulse will be distributed to the nodes of the triangle.

ptPair Point-triangle pair to be impulsed.

void WorldResponser::distributeEEImpulse (const float *Ibar*, const EEPair & *eePair*) [protected]

Function that distributes an impulse, applied to an edge-edge pair, to the nodes.

Parameters:

Ibar Total impulse to be applied to the edges. This impulse will be distributed to the nodes of the edges.
eePair Edge-edge pair to be impulsed.

PositionUpdater WorldResponser::getPositionUpdater () const [protected]

_positionUpdater object accessor.

VelocityUpdater WorldResponser::getVelocityUpdater () const [protected]

_velocityUpdater object accessor.

WorldResponser::VelocitiesMutator & WorldResponser::getVelocitiesMutator () [protected]

_velocityMutator object accessor.

The documentation for this class was generated from the following files:

src/PMDO/PMDO/Responser/WorldResponser.h
src/PMDO/PMDO/Responser/WorldResponser.cpp

5.2.20 WorldResponser::VelocitiesMutator Class Reference

Nested internally used to change the mean velocity of the nodes in the scene.

`#include <WorldResponser.h>`

Public Member Functions

VelocitiesMutator (RCShdPtr< WorldMeanVelocities > worldMeanVelocities)

Constructor of the class.

void addVariation (const Object *object, unsigned int vertexId, const DpoVector &variation)

Function that adds a variation to the number of velotity variations to be applied to a node.

RCShdPtr< WorldMeanVelocities > applyVariations ()

This function applies all the velocity modifications added before, with addVariation .

Detailed Description

Nested internally used to change the mean velocity of the nodes in the scene.

This class first adds the velocity variations and then makes their average and applies them when *applyVariations()* is called.

Constructor & Destructor Documentation

WorldResponser::VelocitiesMutator::VelocitiesMutator (RCShdPtr< WorldMeanVelocities > *worldMeanVelocities*)

Constructor of the class.

Parameters:

worldMeanVelocities Mean velocities of the objects in the scene that will be modified later.

Member Function Documentation

void WorldResponser::VelocitiesMutator::addVariation (const Object * *object*, unsigned int *vertexId*, const DpoVector & *variation*)

Function that adds a variation to the number of velotity variations to be applied to a node.

When *applyVariations()* is called later, all the variations added to that node will be averaged before application.

Parameters:

object Object that contains the node whose velocity will be modified.
vertexId Identification number of the vertex in the mesh of the object.
variation Velocity vector to be added as a modification.

RCShdPtr< WorldMeanVelocities > WorldResponser::VelocitiesMutator::applyVariations ()

This function applies all the velocity modifications added before, with *addVariation* .

Returns:

Modified mean velocities of all the objects in the world.

The documentation for this class was generated from the following files:

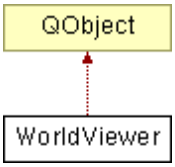
src/PMDO/PMDO/Responser/WorldResponser.h
src/PMDO/PMDO/Responser/WorldResponser.cpp

5.2.21 WorldViewer Class Reference

Upper layer class that contains the two windows, *_explorerWidget* and *_controllerWidget* , that compose the main user interface.

```
#include <WorldViewer.h>
```

Inheritance diagram for WorldViewer:



Public Slots

void playSimulation ()

Qt slot to play the simulation.

void pauseSimulation ()

Qt slot to pause the simulation.

void stepSimulation ()

Qt slot to advance the simulation one step forward.

void rewindSimulation ()
Qt slot to play the reinitialize the scene.
void setSaveIterationMeshes (bool saveIterationMeshes)
Qt slot to save the iteration meshes.

Signals

void updateRenderingModels ()
Qt signal to update the rendered models according to the changes in the _world .
void updateTimeDisplay ()
Qt signal to update the simulated time display, in the _explorerWidget .
void saveRenderChkToggled (bool)
Qt signal to save the rendered scene as an image.

Public Member Functions

WorldViewer (QObject *parent=0, const char *name=0)
Constructor of the class.
World * getWorldPtr ()
World pointer accessor.
void sendUpdateSignal ()
Method that emits updateRenderingModels to start the render updating process.
bool getStepFlag ()
_stepFlag accessor. Used internally.
TimeCallback * getTimeCallback ()
_timeCallback accessor. Used internally.
void emitUpdateTimeDisplay ()
Updates the time displayed so that it matches with _world->_time .

Classes

class TimeCallback
Nested class that holds the time callback information.

Detailed Description

Upper layer class that contains the two windows, *_explorerWidget* and *_controllerWidget* , that compose the main user interface.

This class lets the user mamipulate and observe the state of the world.

Constructor & Destructor Documentation

WorldViewer::WorldViewer (QObject * *parent* = 0, const char * *name* = 0)

Constructor of the class.

This constructor creates the *world* , the *explorerWidget* and the *controllerWidget* , initializing the simulator.

Parameters:

parent Parent Qt Object.

name Name of the Qt Object.

Member Function Documentation

void WorldViewer::updateRenderingModels () [signal]

Qt signal to update the rendered models according to the changes in the *_world* .

void WorldViewer::updateTimeDisplay () [signal]

Qt signal to update the simulated time display, in the *_explorerWidget* .

void WorldViewer::saveRenderChkToggled (bool *t0*) [signal]

Qt signal to save the rendered scene as an image.

void WorldViewer::playSimulation () [slot]

Qt slot to play the simulation.

void WorldViewer::pauseSimulation () [slot]

Qt slot to pause the simulation.

void WorldViewer::stepSimulation () [slot]

Qt slot to advance the simulation one step forward.

void WorldViewer::rewindSimulation () [slot]

Qt slot to play the reinitialize the scene.

void WorldViewer::setSaveIterationMeshes (bool *saveIterationMeshes*) [slot]

Qt slot to save the iteration meshes.

World * WorldViewer::getWorldPtr ()

World pointer accessor.

void WorldViewer::sendUpdateSignal ()

Method that emits *updateRenderingModels* to start the render updating process.

bool WorldViewer::getStepFlag ()

_stepFlag accessor. Used internally.

WorldViewer::TimeCallback * WorldViewer::getTimeCallback ()

_timeCallback accessor. Used internally.

void WorldViewer::emitUpdateTimeDisplay ()

Updates the time displayed so that it matches with *_world->_time* .

The documentation for this class was generated from the following files:

src/PMDO/Viewer/WorldViewer.h
src/PMDO/Viewer/moc_WorldViewer.cpp
src/PMDO/Viewer/WorldViewer.cpp

5.2.22 WorldViewer::TimeCallback Class Reference

Nested class that holds the time callback information.

`#include <WorldViewer.h>`

Public Member Functions

TimeCallback (WorldViewer *worldViewer, float interval)

Constructor. interval is the time interval between calls.

SoTimerSensor & getSensor ()

_renderSensor accessor.

Detailed Description

Nested class that holds the time callback information.

Constructor & Destructor Documentation

WorldViewer::TimeCallback::TimeCallback (WorldViewer * *worldViewer*, float *interval*)

Constructor. *interval* is the time interval between calls.

Member Function Documentation

SoTimerSensor & WorldViewer::TimeCallback::getSensor ()

_renderSensor accessor.

The documentation for this class was generated from the following files:

src/PMDO/Viewer/WorldViewer.h
src/PMDO/Viewer/WorldViewer.cpp