Sistemas Informáticos
Curso 2005/2006

Rigid Body Simulation

David B. Jenkins López
Álvaro del Monte Freitas
Manuel Montenegro Montes

Dirigido por:
Prof. Pedro Jesús Martín de la Calle
Dpto. Sistemas Informáticos y Programación.

Facultad de Informática
Universidad Complutense de Madrid

Rigid Body Simulation

David B. Jenkins López
Álvaro del Monte Freitas
Manuel Montenegro Montes

3 de Julio de 2006

Resumen

Los sólidos rígidos son un caso especial de un sistema de partículas, donde la distancia relativa entre dos puntos cualquiera del sistema permanece constante.

El objetivo de nuestro proyecto es el desarrollo e implementación en lenguaje C++ de una API que permita modelar y simular el comportamiento de los sólidos rígidos. Este comportamiento tiene dos facetas: la correspondiente a cada cuerpo en solitario (translaciones, rotaciones) y la debida al contacto entre varios (colisión y equilibrio). Para ello es necesario aplicar conceptos relacionados con la mecánica (cinemática y dinámica del sólido rígido), el análisis numérico (métodos de resolución de ecuaciones diferenciales ordinarias) y la geometría computacional (detección de colisiones).

Además de la API hemos desarrollado varios ejemplos que muestran diversas capacidades de ésta. Para la visualización de estos ejemplos hemos utilizado la librería gráfica *OpenGL*. No obstante, se ha procurado que la representación gráfica de las escenas simuladas sea completamente independiente de su estructura interna.

El resultado de este proyecto puede ser aplicado a diversos campos, como la programación de videojuegos y simulación de sistemas físicos.

Abstract

Rigid bodies are a special case of particle systems where the relative distance between each pair of points of the system remain constant.

Our project's aim is to develop and provide a C++ implementation of an API which allow us to model and simulate rigid bodies' behaviour, which has two aspects: the one corresponding to each body by itself (translations and rotations) and the one due to contact among bodies (collision and balance). In order to achieve this, concepts related to rigid body mechanics (kinematics and kinetics), numerical analysis (ordinary differential equations solving methods) and computational geometry (collision detection) must be applied.

In addition to this API, several examples have been developed to show its capabilities. *OpenGL* graphics library has been used to render these examples. However, our purpose was to isolate the graphical representation of simulated scenes of its inner structure.

Our project's result can be applied to many areas, like game programming and simulation of physical systems.

Palabras clave: sólido rígido, detección de colisiones, impulso, resting contact, simulación, bounding box.

Índice general

Índice de Figuras

Índice de Algoritmos

Capítulo 1

Dinámica del Sólido Rígido

1.1. Introducción

En física se utiliza el concepto de sólido rígido como idealización de los cuerpos sólidos. En un sólido rígido la distancia entre dos puntos cualesquiera del mismo es siempre la misma, independientemente de las fuerzas que actúen sobre él; es decir, no admite deformaciones. La rama de la física encargada del estudio de este tipo de cuerpos se conoce como *Dinámica del Sólido Rígido*.

1.2. Cinemática del Sólido Rígido

Para poder llevar a cabo la simulación de un objeto debemos almacenar su posición y su orientación. La primera la expresaremos mediante un vector $\mathbf{x}(t)$, que indica la translación del centro de masas del objeto con respecto al origen de coordenadas de la escena.

La representación de una orientación en el espacio no es tan sencilla. En tres dimensiones, la rotación de un objeto tiene tres grados de libertad, por lo que en principio bastarían tres escalares para representarla, lo que se conoce como los *ángulos de Euler*. Expresamos cada rotación mediante tres ángulos (roll, pitch, yaw). Esta representación resulta muy intuitiva, fácil de visualizar y óptima en espacio. Sin embargo, el modelado de las ecuaciones de movimiento utilizando esta representación conlleva ciertos problemas: uno de ellos es la aparición de cosenos en el denominador de la ecuación, lo cual implica el tratamiento de casos "especiales" cuando el ángulo correspondiente es $\pi/2$ ó $-\pi/2$ para evitar la división por cero.

Otra forma que tenemos para expresar las rotaciones es utilizando una matriz de 3x3, llamada *matriz de rotación*, $\mathbf{R}(t)$. La forma geométrica de un sólido rígido puede definirse de

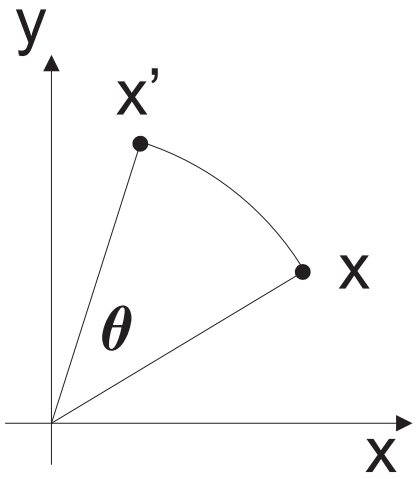


Figura 1.1: Rotación alrededor del eje z

forma invariable en un *sistema de coordenadas local*. Si tenemos la descripción geométrica de un objeto en dicho sistema local, podemos transformarla al sistema de coordenadas *global* con ayuda de $\mathbf{x}(t)$ y $\mathbf{R}(t)$. En efecto, las coordenadas de un punto en el sistema de coordenadas local ($\mathbf{p}$) y en el sistema de coordenadas global ($\mathbf{p'}$) quedan relacionadas mediante la siguiente expresión:

$$\mathbf{p'} = \mathbf{R}(t)\ \mathbf{p} + \mathbf{x}(t) \tag{1.1}$$

Para realizar, por ejemplo, una rotación sobre el eje z (Figura 1.1) utilizaríamos la matriz:

$$\mathbf{R} = \begin{pmatrix} cos\ \theta & -sen\ \theta & 0 \\ sen\ \theta & cos\ \theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Las matrices de rotación son ortogonales, es decir, su determinante es 1, y al multiplicarlas por su transpuesta obtenemos la matriz identidad ($\mathbf{R}\mathbf{R}^T = \mathbf{1}$). Es importante conservar esta propiedad de ortogonalidad durante la simulación, ya que a medida que se realizan operaciones con las matrices pueden aparecer errores de redondeo, que pueden provocar que el determinante se aleje de 1. En este caso, la matriz no especificaría sólo una rotación: podría también transladar y escalar los objetos. Otra desventaja de esta representación es que se necesitan nueve escalares para almacenar una rotación. Los quaterniones (Sección 1.6) ofrecen una representación más eficiente.

Para modelar el movimiento y el giro de los objetos necesitamos otras dos magnitudes: La *velocidad lineal* se define como:

$$\mathbf{v} = \frac{d\mathbf{x}}{dt} \tag{1.2}$$

Para representar el giro utilizamos la *velocidad angular*, ω. La velocidad angular es un vector. La dirección del vector muestra el eje alrededor del cual el objeto está girando, mientras que su módulo indica la rapidez del giro. La velocidad angular se relaciona con la matriz de rotación mediante la siguiente expresión:

$$\frac{d\mathbf{R}}{dt} = \Omega\mathbf{R} \tag{1.3}$$

Donde:

$$\Omega = \begin{pmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{pmatrix}$$

La derivación de esta ecuación se describe detalladamente en [Bar97c].

Una propiedad interesante de la matriz Ω es que para cualquier vector $\mathbf{v}$:

$$\Omega\mathbf{v} = \omega \times \mathbf{v} \tag{1.4}$$

En efecto:

$$\Omega\mathbf{v} = \begin{pmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = \begin{pmatrix} \omega_y v_z - \omega_z v_y \\ \omega_z v_x - \omega_x v_z \\ \omega_x v_y - \omega_y v_x \end{pmatrix} = \omega \times \mathbf{v}$$

En ciertas ocasiones será necesario calcular la velocidad lineal en un determinado punto $\mathbf{p}$ (en el sistema de coordenadas local) del sólido rígido en movimiento. Derivando a ambos lados de la ecuación (1.1) obtenemos:

$$\frac{d\mathbf{p'}}{dt} = \frac{d\mathbf{R}}{dt}\mathbf{p} + \frac{d\mathbf{x}}{dt}$$

Utilizando las ecuaciones (1.2) y (1.3) tenemos:

$$\begin{aligned} \mathbf{v'} &= \Omega\ \mathbf{R}\ \mathbf{p} + \mathbf{v} \\ &= \Omega\ (\mathbf{R}\ \mathbf{p} + \mathbf{x} - \mathbf{x}) + \mathbf{v} \\ &= \Omega\ (\mathbf{p'} - \mathbf{x}) + \mathbf{v} \\ &= \omega \times (\mathbf{p'} - \mathbf{x}) + \mathbf{v} \end{aligned}$$

Donde $\mathbf{v'}$ indica la velocidad del punto $\mathbf{p'}$ (en el sistema de coordenadas global) y $\mathbf{v}$ la velocidad del centro de masas del cuerpo. La expresión para $\mathbf{v'}$ se separa, por tanto, en dos componentes: una lineal ($\mathbf{v}$) y otra angular $\omega \times (\mathbf{p'} - \mathbf{x})$.

1.3. Propiedades de masa del Sólido Rígido

Las propiedades de masa de un sólido rígido son importantes en el estudio del comportamiento de este tipo de cuerpos, ya que la velocidad (tanto lineal como angular) y la respuesta a la aplicación de una fuerza están en función de dichas propiedades.

En primer lugar, la *masa* de un objeto es la medida que indica la resistencia que ofrece un cuerpo a ser desplazado. Cuanto mayor es la masa, más difícil será cambiar su estado de movimiento. La masa total de un cuerpo podría calcularse sumando las masas de todas las partículas elementales que lo conforman; para cada partícula elemental se calculará su masa multiplicando su densidad (ρ) por su volúmen. Tendríamos una expresión de la forma:

$$m = \int \rho \, dV$$

En el caso de cuerpos con densidad uniforme, la expresión sería $m = \rho V$. No obstante, en la API *RBS* **la masa de un cuerpo se supondrá ya especificada por el programador**, por lo que no se procederá a su cálculo en ningún caso.

Otro concepto importante es el *centro de gravedad* de un sólido rígido, que es el punto dentro del mismo alrededor del cual se encuentra la masa uniformemente distribuída. En términos de dinámica de rotación, es el punto en el que puede actuar cualquier fuerza sin causar una rotación en el sólido. Este punto lo especificamos en el sistema de coordenadas global de la escena; para su cálculo habría que dividir el sólido en infinitas masas elementales, cada una con su centro, para posteriormente multiplicar para cada una su centro y su masa, sumarlas todas y dividir el resultado entre la masa total del sólido. Esto conllevaría el cálculo de integrales sobre el volúmen del cuerpo, al igual que en la ecuación anterior. No obstante, en nuestro caso podemos dividir el sólido en una cantidad n finita de masas elementales, obteniendo la siguiente expresión:

$$\mathbf{CM} = \frac{(\sum_{i=1}^{n} x_i m_i \ , \sum_{i=1}^{n} y_i m_i \ , \sum_{i=1}^{n} z_i m_i)}{\sum_{i=1}^{n} m_i} \tag{1.5}$$

En el caso de las formas geométricas básicas, el centro de masas suele ser fácil de calcular (Figura 1.2). No obstante, cuando se manejan **mallas de polígonos** que describen una forma arbitraria consideraremos que se trata de un sistema de partículas en el que la masa total del cuerpo se encuentra repartida uniformemente entre los distintos vértices que la componen. En estos casos se utilizará la Ecuación 1.5 para calcular el centro de masas.

Otra propiedad de relevancia para el movimiento de los cuerpos es su *momento de inercia*. El momento de inercia mide la distribución radial de la masa de un cuerpo a lo largo de un eje de rotación. Al igual que la masa indicaba la resistencia de un objeto al movimiento *lineal*, el momento de inercia indica la resistencia de un objeto al movimiento *rotacional*, pero a diferencia de la anterior, ésta varía según el eje de rotación que consideremos.

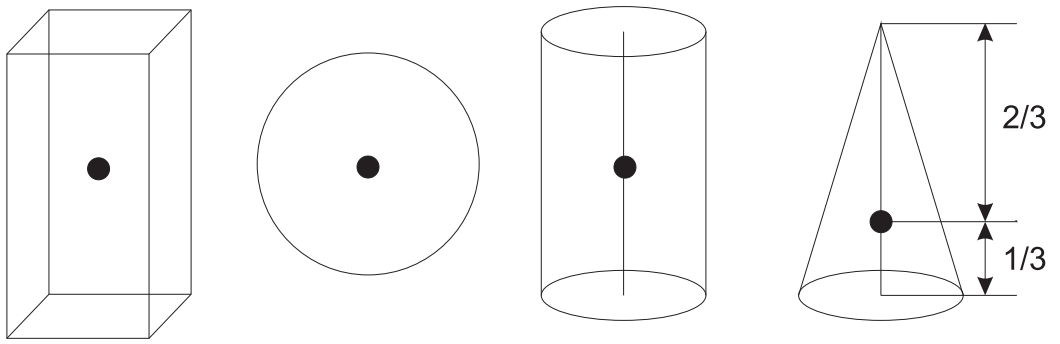


Figura 1.2: Posición del centro de masas para formas geométricas básicas.

Dado que el valor del momento de inercia depende del eje de rotación que se considere, debemos expresar el momento de inercia de un cuerpo mediante un *tensor*. Un tensor, al igual que un vector, es una expresión matemática que tiene magnitud y dirección. Sin embargo, a diferencia del vector, la magnitud no es única, sino que depende de la dirección. En particular, el momento de inercia de un cuerpo es un tensor de segundo orden, y puede formularse en términos de una matriz 3x3:

$$\mathbf{I} = \begin{pmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{yx} & I_{yy} & -I_{yz} \\ -I_{zx} & -I_{zy} & I_{zz} \end{pmatrix}$$

Donde:

$$I_{xx} = \int (y^2 + z^2)\, dV \qquad I_{xy} = I_{yx} = \int xy\, dV$$
$$I_{yy} = \int (z^2 + x^2)\, dV \qquad I_{yz} = I_{zy} = \int yz\, dV$$
$$I_{zz} = \int (x^2 + y^2)\, dV \qquad I_{xz} = I_{zx} = \int zx\, dV$$

Cabe destacar que para cualquier cuerpo existe una orientación tal que $\mathbf{I}$ es una matriz diagonal. Si el cuerpo a tratar tiene una geometría simple (esfera, parelelepípedo, cono, cilindro), podemos calcularla fácilmente mediante las fórmulas resultantes de resolver cada integral (Figura 1.3). Estas fórmulas se utilizarán en *RBS* para las **geometrías básicas**.

En el caso de las mallas con geometría arbitraria, tendremos que utilizar métodos numéricos para el cálculo de $\mathbf{I}$. Un método sencillo consiste en discretizar el sólido delimitado por la malla en un sistema de n partículas. Suponiendo que $\mathbf{r}_i$ es la posición de

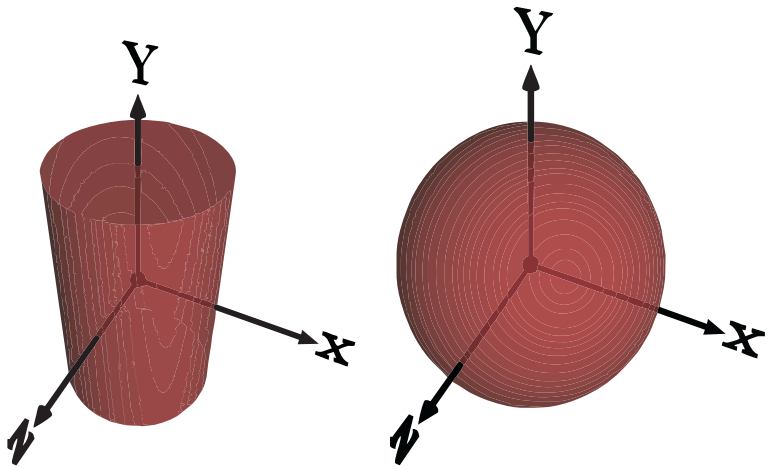


Figura 1.3: Tensor de inercia para el cilindro: $I_{xx} = I_{zz} = (1/4)mr^2 + (1/12)ml^2$ $I_{yy} = (1/2)mr^2$. Tensor de inercia para la esfera: $I_{xx} = I_{yy} = I_{zz} = (2/5)mr^2$

la i-ésima partícula con respecto al centro de masas del cuerpo y m_i es su masa, podemos expresar su tensor de inercia como:

$$\mathbf{I} = \sum_{i=1}^{n} \begin{pmatrix} m_i(r_{iy}^2 + r_{iz}^2) & -m_i r_{ix} r_{iy} & -m_i r_{ix} r_{iz} \\ -m_i r_{iy} r_{ix} & m_i(r_{ix}^2 + r_{iz}^2) & -m_i r_{iy} r_{iz} \\ -m_i r_{iz} r_{ix} & -m_i r_{iz} r_{iy} & m_i(r_{ix}^2 + r_{iy}^2) \end{pmatrix} \tag{1.6}$$

Para discretizar el sólido en un sistema de partículas tendremos que crear una caja que lo delimite y tomar muestras dentro de la caja. Cuando una muestra cae dentro del sólido (acierto) añadimos la matriz correspondiente a la matriz que tenemos acumulada. La descripción de este proceso podemos encontrarla en el Algoritmo 1.1.

Éste es el método utilizado en *RBS*para las **mallas de polígonos**. Otros métodos más precisos para el cálculo del tensor de inercia, basados en el teorema de Green ([Kap91]) pueden encontrarse en [Mir96a].

1.4. Dinámica del Sólido Rígido

Los responsables de causar cambios en la velocidad de un objeto son las *fuerzas*. Según la Primera Ley de Newton, todo cuerpo persevera en su estado de reposo o movimiento a menos que la acción de una fuerza le obligue a cambiarlo.

En el estudio de un sólido rígido se realiza distinción entre *fuerza* ($\mathbf{F}$) y *torque* (τ). La primera es la responsable del movimiento lineal de una partícula, mientras que el segundo

$\mathbf{I} \leftarrow \mathbf{0}$; $aciertos \leftarrow 0$
Calcular la caja delimitadora del poliedro
para cada muestra (x, y, z) tomada dentro de la caja delimitadora:
  si (x, y, z) se encuentra dentro del poliedro **entonces**

$$\mathbf{I} \leftarrow \mathbf{I} + \begin{pmatrix} y^2+z^2 & -x*y & -x*z \\ -x*y & x^2+z^2 & -y*z \\ -x*z & -y*z & x^2+y^2 \end{pmatrix}$$

    $aciertos \leftarrow aciertos + 1$
  fsi
fpara
$\mathbf{I} \leftarrow \mathbf{I} * (m/aciertos)$

Algoritmo 1.1: Cálculo del tensor de inercia mediante toma de muestras

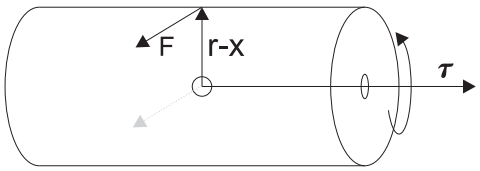


Figura 1.4: Rotación provocada por una fuerza

es el responsable del movimiento rotacional y se expresa en términos de $\mathbf{F}$. El torque se define como:

$$\tau = (\mathbf{r} - \mathbf{x}) \times \mathbf{F} \tag{1.7}$$

Donde $\mathbf{r}$ es el punto de aplicación de la fuerza (en el sistema de coordenadas global) y $\mathbf{x}$ es la posición del centro de masas del objeto. Por tanto, cuanto mayor es la distancia del punto de aplicación al centro de masas, mayor será el torque y viceversa. Además, la dirección del torque es la misma que la dirección de la velocidad angular que éste provoca (Figura 1.4).

Cuando en un cuerpo actúan varias fuerzas $\mathbf{F}_i$, la fuerza externa total $\mathbf{F}$ es la suma de todas ellas. Ánalogamente, el torque externo total es la suma de todos los torques.

El *momento lineal* de una partícula se define como el producto de la masa por su velocidad ($\mathbf{p}_i = m_i\mathbf{v}_i$). En el caso de un sistema de partículas, el momento total del sistema es igual a la suma del momento de todas las partículas que lo forman.

$$\mathbf{P} = \sum_{i=1}^{n} \mathbf{P}_i$$

Un resultado importante es que el momento lineal total de un sólido rígido es el mismo que el momento lineal que tendría una partícula con la misma masa y velocidad que el mismo.

$$\mathbf{P} = M\mathbf{v} \tag{1.8}$$

Ahora bien, necesitamos una ecuación que relacione la fuerza total externa aplicada sobre un objeto y el cambio de velocidad lineal que éste sufre a consecuencia. Ésta relación viene dada por la *Segunda Ley de Newton*:

$$\boxed{\mathbf{F} = \frac{d\mathbf{P}}{dt}} \tag{1.9}$$

Para describir el cambio de velocidad angular de un objeto necesitamos una magnitud análoga al momento lineal. Esta magnitud es el *momento angular* ($\mathbf{L}$), que viene definido por la ecuación:

$$\mathbf{L} = \mathbf{I}\omega \tag{1.10}$$

Y del mismo modo que la fuerza total provoca variación en el momento lineal, es el torque quién provoca variación en el momento angular:

$$\boxed{\tau = \frac{d\mathbf{L}}{dt}} \tag{1.11}$$

1.5. Simulación de Sólidos Rígidos independientes

En *RBS* la simulación se realiza por *ciclos*. Existe un estado para cada objeto, que indica su posición, orientación, etc. Al principio de cada ciclo de simulación se comprueban las fuerzas que actúan sobre él, y los torques correspondientes. Posteriormente se actualiza el estado de los objetos atendiendo a estas fuerzas, con lo que finaliza el ciclo.

Las magnitudes que se almacenan para cada objeto (y por tanto definen su estado) son:

- Posición
- Orientación
- Momento lineal
- Momento angular

Estas cuatro magnitudes conforman el *vector de estado* $\mathbf{Y}$:

$$\mathbf{Y}(t) = \begin{pmatrix} \mathbf{x}(t) \\ \mathbf{R}(t) \\ \mathbf{P}(t) \\ \mathbf{L}(t) \end{pmatrix} \tag{1.12}$$

Ahora bien, para cada paso de simulación necesitaremos saber la *variación* de dicho estado:

$$\frac{d\mathbf{Y}(t)}{dt} = \begin{pmatrix} d\mathbf{x}(t)/dt \\ d\mathbf{R}(t)/dt \\ d\mathbf{P}(t)/dt \\ d\mathbf{L}(t)/dt \end{pmatrix} = \begin{pmatrix} \mathbf{v}(t) \\ \Omega(t)\mathbf{R}(t) \\ \mathbf{F}(t) \\ \tau(t) \end{pmatrix} \tag{1.13}$$

¿Podemos conocer la variación a partir de nuestra representación del estado? La respuesta es sí, ya que:

- La fuerza total ($\mathbf{F}$) la podemos conocer sumando las fuerzas individuales que actúan sobre nuestro objeto, y éstas vienen especificadas por el usuario de la API.
- El torque (τ) se puede calcular mediante su definición a partir de la ecuación (1.7).
- La velocidad lineal podemos despejarla de la ecuación (1.8), sin más que despejar $\mathbf{v}$:

$$\mathbf{v}(t) = \frac{\mathbf{P}(t)}{M}$$

- La matriz Ω está en función de la velocidad angular ω. Esta última puede conseguirse despejando a partir de la definición del momento angular (1.10) :

$$\omega(t) = \mathbf{I}^{-1}(t)\mathbf{L}(t)$$

Con respecto al cálculo de $\mathbf{I}^{-1}$, hay que tener en cuenta que se trata de la matriz inversa del tensor de inercia asociado a la *orientación actual* del objeto. Los métodos vistos en la sección 1.3 calculaban $\mathbf{I}$ atendiendo al sistema de coordenadas *local* del objeto, por lo que habría que recalcular $\mathbf{I}$ en cada paso de simulación para poder utilizarlo en la ecuación anterior. Esto resultaría muy costoso desde el punto de vista de la eficiencia. Afortunadamente, se puede deducir una ecuación que transforma el tensor de inercia expresado en coordenadas locales al expresado en coordenadas globales:

$$\mathbf{I}(t) = \mathbf{R}(t)\mathbf{I}_0\mathbf{R}(t)^T$$

Donde $\mathbf{I}_0$ es el tensor de inercia expresado en coordenadas locales al objeto, que es el que se calculará al principio de la simulación utilizando lo comentado en la sección 1.3. Podemos aprovechar la propiedad de ortogonalidad de $\mathbf{R}$ para deducir:

$$\begin{aligned}\mathbf{I}^{-1}(t) &= (\mathbf{R}(t)\ \mathbf{I}_0\ \mathbf{R}(t)^T)^{-1} \\ &= (\mathbf{R}(t)^T)^{-1}\ \mathbf{I}_0^{-1}\ (\mathbf{R}(t))^{-1} \\ &= \mathbf{R}(t)\ \mathbf{I}_0^{-1}\ \mathbf{R}(t)^T\end{aligned}$$

El cálculo del nuevo estado **Y'** a partir del estado inicial $\mathbf{Y}$ y de $d\mathbf{Y}/dt$ requiere la utilización de métodos numéricos para el cálculo de ecuaciones diferenciales, que se verán en el Capítulo 2.

1.6. Quaterniones

En la Sección 1.2 hemos visto que la orientación de un objeto en el espacio podía representarse mediante ángulos de Euler o matrices de rotación. A continuación veremos otra representación que también puede utilizarse para describir rotaciones: los quaterniones.

Los quaterniones son una extensión de los números reales, del mismo modo que lo son los números complejos. Mientras que en éstos últimos se contaba con una unidad imaginaria i que cumplía $i^2 = -1$, en los quaterniones disponemos de varias unidades imaginarias: i, j y k, que cumplen:

$$i^2 = j^2 = k^2 = ijk = -1$$

Un quaternion $\mathbf{q}$ suele escribirse en la forma:

$$\mathbf{q} = q_n + q_x i + q_y j + q_z k \qquad q_n, q_x, q_y, q_z \in \mathbb{R}$$

Aunque también puede escribirse en la forma $[q_n, \mathbf{v}]$, donde $\mathbf{v}$ es un vector con tres componentes (las q_x, q_y y q_z). Los quaterniones permiten varias operaciones como la suma y multiplicación (ver Figura 1.5).

Como hemos dicho anteriormente, un quaternión puede utilizarse para describir una rotación en el espacio. En este sentido consideramos un vector unitario $\mathbf{v}$ que representa un eje de rotación arbitrario. Para expresar una rotación θ alrededor de dicho eje el quaternión correspondiente queda de la siguiente forma:

$$\mathbf{q} = \left[cos\left(\frac{\theta}{2}\right), sin\left(\frac{\theta}{2}\right)\mathbf{v} \right]$$

Como $|\mathbf{v}| = 1$, se puede demostrar fácilmente que $|\mathbf{q}| = 1$.

Dados $\mathbf{p} = [n_p, \mathbf{v}_p], \mathbf{q} = [n_q, \mathbf{v}_q], \mathbf{v}_p = (p_x, p_y, p_z), \mathbf{v}_q = (q_x, q_y, q_z)$:

Suma:	$\mathbf{q} + \mathbf{p} = [n_q + n_p, (x_q + x_p)i + (y_q + y_p)j + (z_q + z_p)k]$
Producto por escalar:	$\lambda\mathbf{q} = [\lambda n_q, \lambda\mathbf{v}_q]$
Producto:	$\mathbf{q} \cdot \mathbf{p} = [n_q n_p - \mathbf{v}_q \cdot \mathbf{v}_p, n_q\mathbf{v}_p + n_p\mathbf{v}_q + (\mathbf{v}_q \times \mathbf{v}_p)]$
Conjugado:	$\bar{\mathbf{q}} = [n_q, -\mathbf{v}_q]$
Módulo:	$\lvert\mathbf{q}\rvert = \sqrt{n_q^2 + q_x^2 + q_y^2 + q_z^2}$
Inversa:	$\mathbf{q}^{-1} = \frac{\bar{\mathbf{q}}}{\mathbf{q}\cdot\bar{\mathbf{q}}}$

Figura 1.5: Operaciones con quaterniones

Al contrario que con las matrices de rotación, aquí tenemos sólo cuatro parámetros en lugar de nueve; por otro lado los quaterniones resultan mucho más fáciles de normalizar que las matrices de rotación. Esto hace que sean la opción preferida a la hora de representar rotaciones en el espacio. Para rotar un punto basta con multiplicar por el quaternión a la izquierda y por su conjugado a la derecha:

$$\mathbf{p'} = \mathbf{q}\mathbf{p}\bar{\mathbf{q}} \tag{1.14}$$

Recordemos que si hubiésemos utilizado matrices de rotación, tendríamos que aplicar:

$$\mathbf{p'} = \mathbf{R}\mathbf{p} \tag{1.15}$$

Combinando (1.14) y (1.15) podemos obtener la siguiente fórmula que permite calcular la matriz de rotación equivalente a un quaternión:

$$\mathbf{R} = \begin{pmatrix} 1 - 2(q_y^2 + q_z^2) & 2q_xq_y - 2n_qq_z & 2n_qq_y + 2q_xq_z \\ 2q_xq_y + 2n_qq_z & 1 - 2(q_x^2 + q_z^2) & -2n_qq_x + 2q_yq_z \\ -2n_qq_y + 2q_xq_z & 2n_qq_x + 2q_yq_z & 1 - 2(q_x^2 + q_y^2) \end{pmatrix} \tag{1.16}$$

También nos interesará transformar la matriz de rotación a quaternión, es decir, la transformación inversa. Sea m la matriz de rotación de origen cuyos elementos se distribuyen de la siguiente forma:

$$\begin{pmatrix} m[0] & m[4] & m[8] & m[12] \\ m[1] & m[5] & m[9] & m[13] \\ m[2] & m[6] & m[10] & m[14] \\ m[3] & m[7] & m[11] & m[15] \end{pmatrix}$$

y q el quaternión que queremos obtener. El proceso se muestra en el Algoritmo 1.2.

Una vez definidos los quaterniones debemos incorporarlos al vector de estado, que queda

si(m[0] > m[5] ∧ m[0] > m[10])
    S = $\sqrt{1{,}0 + m[0] - m[5] - m[10]}$ * 2;
    q.x = 0.25 * S;
    q.y = (m[1] + m[4]) / S;
    q.z = (m[8] + m[2]) / S;
    q.n = (m[6] - m[9]) / S;
sino si(m[5] > m[10])
    S = $\sqrt{1{,}0 + m[5] - m[0] - m[10]}$ * 2;
    q.x = (m[1] + m[4]) / S;
    q.y = 0.25 * S;
    q.z = (m[6] + m[9]) / S;
    q.n = (m[8] - m[2]) / S;
sino
    S = $\sqrt{1{,}0 + m[10] - m[0] - m[5]}$ * 2;
    q.x = (m[8] + m[2]) / S;
    q.y = (m[6] + m[9]) / S;
    q.z = 0.25 * S;
    q.n = (m[1] - m[4]) / S;

Algoritmo 1.2: Algoritmo de transformación de la matriz de rotación m al quaternión q.

de la siguiente forma:

$$\mathbf{Y}(t) = \begin{pmatrix} \mathbf{x}(t) \\ \mathbf{q}(t) \\ \mathbf{P}(t) \\ \mathbf{L}(t) \end{pmatrix} \tag{1.17}$$

El problema consiste ahora en la obtención de una expresión para la derivada de $\mathbf{q}(t)$, para obtener el siguiente estado. Supongamos que el cuerpo está rotando a una velocidad angular constante de $\omega(t)$. Tras un lapso de tiempo Δt la rotación producida será:

$$\left[cos\left(\frac{\theta}{2}\right), sin\left(\frac{\theta}{2}\right)\mathbf{v}\right] = \left[cos\left(\frac{|\omega(t)|\Delta t}{2}\right), sin\left(\frac{|\omega(t)|\Delta t}{2}\right)\frac{\omega(t)}{|\omega(t)|}\right]$$

Ahora bien, el objeto tenía una orientación inicial ($\mathbf{q}(t_0)$). Combinando la nueva rotación con dicha orientación tenemos:

$$\mathbf{q}(t_0 + \Delta t) = \left[cos\left(\frac{|\omega(t_0)|\Delta t}{2}\right), sin\left(\frac{|\omega(t_0)|\Delta t}{2}\right)\frac{\omega(t_0)}{|\omega(t_0)|}\right]\mathbf{q}(t_0)$$

Al hacer $t = t_0 + \Delta t$:

$$\mathbf{q}(t) = \left[cos\left(\frac{|\omega(t_0)|(t - t_0)}{2}\right), sin\left(\frac{|\omega(t_0)|(t - t_0)}{2}\right)\frac{\omega(t_0)}{|\omega(t_0)|}\right]\mathbf{q}(t_0)$$

Lo cual resulta, tomando la derivada con respecto a t a ambos lados de la ecuación (ver [Bar97c] para la derivación completa):

$$\boxed{\frac{d\mathbf{q}}{dt} = \frac{1}{2}\omega\mathbf{q}} \tag{1.18}$$

Donde ω representa realmente al quaternion $[0, \omega]$

1.7. Detalles de implementación

En esta sección se describirá la parte de la interfaz *RBS* encargada de modelar los objetos que conforman una escena, sus propiedades de masa, las fuerzas que actúan sobre ellos y los efectos producidos por dichas fuerzas.

1.7.1. Elementos matemáticos básicos

En primer lugar implementamos las clases que representan elementos matemáticos tales como vectores, matrices y quaterniones. Estas clases serán utilizadas en toda la API.

Se implementaron las siguientes clases:

- **rbsVector3**: Las instancias de esta clase representan vectores en el espacio. Además de permitir el acceso a cada una de las tres componentes por separado, proporciona operaciones sobre estos: suma, resta, multiplicación por un escalar, producto escalar, producto vectorial, etc.

- **rbsMatriz3x3**: Encapsula una matriz de tres filas y tres columnas. No obstante, cabe destacar que por motivos de eficiencia a la hora de realizar la representación gráfica de una escena con OpenGL, las matrices están internamente representadas como vectores de 16 elementos. La disposición de elementos en el vector se encuentra representada de la siguiente forma:

$$\begin{pmatrix} m[0] & m[4] & m[8] & m[12]=0 \\ m[1] & m[5] & m[9] & m[13]=0 \\ m[2] & m[6] & m[10] & m[14]=0 \\ m[3]=0 & m[7]=0 & m[11]=0 & m[15]=1 \end{pmatrix}$$

  Siendo m un vector de 16 elementos. La clase implementa además las operaciones habituales sobre las matrices: suma, resta, producto por un escalar, producto por otra matriz, cálculo de la inversa, cálculo del determinante, etc.

- **rbsQuaternion**: Los objetos de esta clase agrupan un componente escalar y uno vectorial (**rbsVector3**). Además del acceso a cada una de estas componentes, permite operaciones tales como la suma, resta, producto, conjugado, etc.

1.7.2. Estructura de la escena

El usuario de la API deberá en primer lugar crear los distintos objetos cuyo comportamiento se quiere simular. Posteriormente se deben añadir a una instancia de la clase **rbsEscena**. Ésta se encargará de agrupar los objetos y de avanzar la simulación cuando se requiera.

Por tanto, podemos considerar la escena como un contenedor de instancias de la clase **rbsObjeto**. Una escena puede contener varios objetos pero un objeto sólo puede formar parte de una escena. **rbsObjeto** es una clase abstracta y, por tanto, no se puede instanciar directamente. Las instancias deben crearse de una de sus tres subclases:

- **rbsParticula**: Las instancias de ésta clase no guardarán ninguna información sobre el comportamiento rotacional. Esto es, sólo pueden desplazarse pero no pueden rotar. Se corresponde con la idealización en Mecánica de una partícula.

- **rbsRotacional**: Se comportan de manera contraria a las partículas. Pueden rotar, pero no desplazarse.

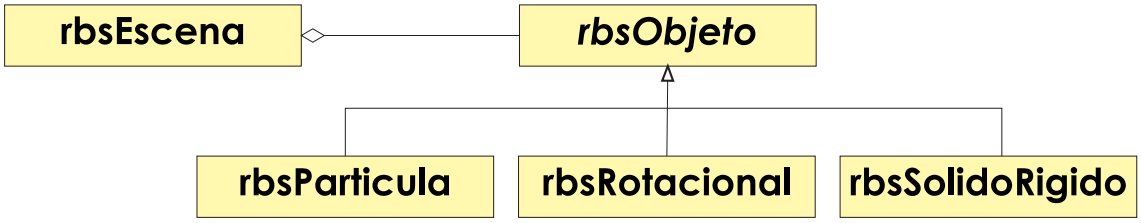


Figura 1.6: Componentes de la escena

`rbsEscena::rbsEscena()`

Constructor de la clase.

`void rbsEscena::InsertarObjeto(rbsObjeto *)`

Añade un objeto nuevo a la escena.

`void rbsEscena::Eliminar(rbsObjeto *)`

Elimina un objeto de la escena.

`TIPOREAL rbsEscena::GetTiempo()`

Obtiene el tiempo de simulación actual.

`TIPOREAL rbsEscena::SiguienteEstado(TIPOREAL tiempo)`

Avanza la simulación una determinada cantidad de tiempo.

Cuadro 1.1: Métodos más importantes de la clase **rbsEscena**

- **rbsSolidoRigido**: Agrupan las propiedades de las dos clases anteriores: Pueden desplazarse y rotar. Guardan información sobre el comportamiento lineal (desplazamiento) y el rotacional.

Los comportamientos lineal y rotacional podrán activarse y desactivarse posteriormente mediante los correspondientes métodos de **rbsObjeto**. No obstante, no podrán activarse los comportamientos para los que el objeto no haya sido diseñado. Por ejemplo, no se puede activar el comportamiento rotacional de una partícula ni el comportamiento lineal de un objeto rotacional.

1.7.3. Propiedades de masa de un objeto

Las propiedades de masa de un objeto comentadas en la Sección 1.3 se encuentran separadas de la clase **rbsObjeto**. En particular, el tensor de inercia y la masa forman

`void rbsObjeto::AplicarFuerza(rbsFuerza *fuerza)`

Añade una fuerza más a la lista de fuerzas del objeto.

`const rbsVector3 &rbsObjeto::GetPosicion() const`

`const rbsRotacion *rbsObjeto::GetRotacion() const`

Obtiene la posición y orientación actuales del objeto.

`const rbsGeometria *rbsObjeto::GetGeometria() const`

Obtiene una referencia constante a la geometría asociada al objeto.

`rbsComportamientoLineal *rbsObjeto::GetComportamientoLineal()`

Obtiene el comportamiento lineal del objeto, esto es, el que varía su posición. En instancias de **rbsRotacional** devuelve $NULL$.

`rbsComportamientoRotacional *rbsObjeto::GetComportamientoRotacional()`

Obtiene el comportamiento rotacional del objeto, esto es, el que varía su orientación. En instancias de **rbsParticula** devuelve $NULL$.

`void rbsObjeto::SetComportamientoRotacionalActivo(bool activo)`

`void rbsObjeto::SetComportamientoLinealActivo(bool activo)`

Activa/Desactiva cada uno de los comportamientos por separado.

`void rbsObjeto::ApilarEstado()`

Guarda el estado actual del objeto en la pila de estados asociada a éste.

`void rbsObjeto::SetComportamientoLinealActivo(bool activo)`

Restaura el estado que se encuentra en la cima de la pila asociada al objeto.

`rbsVector3 rbsObjeto::VelocidadPunto(const rbsVector3 &punto) const`

Calcula la velocidad lineal de un determinado punto del objeto.

Cuadro 1.2: Métodos más importantes de la clase **rbsObjeto**

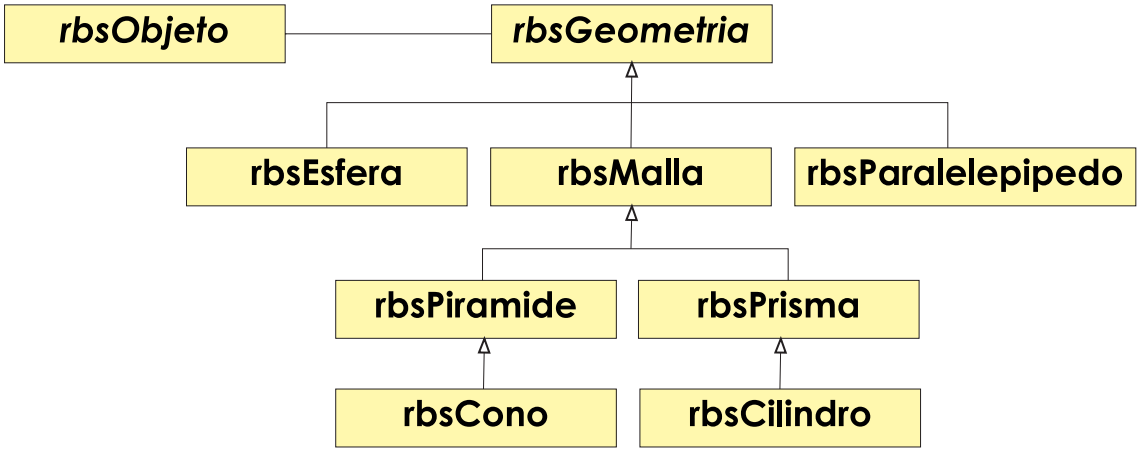


Figura 1.7: Jerarquía de la clase **rbsGeometria**

`const TIPOREAL &rbsGeometria::GetMasa() const`

Acceso a la masa del objeto

`const rbsMatriz3x3 &rbsGeometria::GetTensorInercia(rbsRotacion*)`

`const rbsMatriz3x3 &rbsGeometria::GetTensorInerciaInv(rbsRotacion*)`

Acceso al tensor de inercia del objeto (y su inverso) a partir de su orientación.

Cuadro 1.3: Métodos relevantes de la clase **rbsGeometria**

parte de la clase rbsGeometria. Un objeto siempre contiene una referencia a una instancia de esta clase, que a su vez puede ser compartida con más objetos.

Hay distintos tipos de geometría: esferas, conos, cilindros, mallas, etc. (Figura 1.7) Cada instancia de **rbsGeometria** contiene además información específica de cada geometría. Por ejemplo, de la geometría **rbsEsfera** se guarda el radio. La finalidad de estas propiedades es, por una parte, el dibujo por pantalla y por otro lado, servir de entrada al algoritmo de cálculo de colisiones correspondiente.

1.7.4. Estado de un objeto

Hemos visto anteriormente que el estado de un objeto estaba formado por cuatro componentes: posición, momento lineal, rotación y momento angular. Las dos primeras forman parte del *comportamiento lineal* de un objeto, mientras que las dos segundas corresponden a su *comportamiento rotacional*.

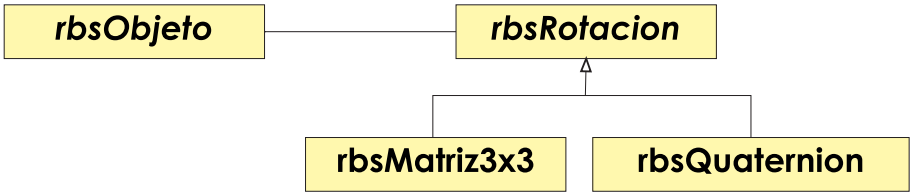


Figura 1.8: Modos de especificar una rotación

La posición de un objeto la podemos representar fácilmente mediante un **rbsVector3**. Sin embargo, en la rotación tenemos dos alternativas posibles: **rbsMatriz3x3** y **rbsQuaternion**. Por defecto se ha escogido la representación del quaternión, aunque el usuario puede especificar el uso de una matriz 3x3 mediante el método `SetModoRotacion()`. De hecho, se tiene una clase abstracta **rbsRotacion** superclase de estas dos representaciones, que especifica los métodos que deben implementar las dos subclases. Contiene métodos para la transformación a ángulos de Euler y viceversa, transformación de un punto, cálculo de la rotación inversa y normalización.

Por otro lado, las clases encargadas de la modificación del estado de la escena son **rbsComportamientoLineal** y **rbsComportamientoRotacional**. Contienen, respectivamente, el momento lineal y el momento angular del objeto y proporcionan métodos de acceso a éstos. Permiten además establecer el umbral mínimo de momento por debajo del cual el objeto se considera parado (método `SetMomentoMinimo()`).

A medida que transcurre la simulación, las rotaciones pueden *desnormalizarse* como consecuencia de errores de redondeo. La matriz 3x3 puede dejar de especificar una rotación y el quaternión puede dejar de tener módulo 1. Para evitar esto se realizan normalizaciones regulares de la orientación actual del objeto. El número de pasos de simulación que deben transcurrir entre normalización y normalización se especifica en el método `SetIntervaloNormalizacion()` del comportamiento rotacional.

Por último, el comportamiento lineal y rotacional de un objeto hacen uso de una instancia de la clase **rbsEDO**, encargada de la resolución de ecuaciones diferenciales ordinarias. Ésto se verá en el Capítulo 2.

1.7.5. Fuerzas aplicadas a un objeto

Como se ha visto en la sección 1.5, para poder determinar el siguiente estado de los objetos es necesario conocer las fuerzas que actúan sobre él en un determinado instante de tiempo. Por ello será necesario disponer de métodos que nos permitan aplicar fuerzas a objetos y determinar las fuerzas a las que está sometido un objeto.

Aplicación de una fuerza

Cuando se define una fuerza será necesario indicar:

- El **objeto** al que afecta.
- El **intervalo de tiempo** en el que actúa.
- La función que determina su **módulo y dirección**.
- La función que determina el **punto de aplicación** sobre el objeto.

En una primera aproximación el usuario de la API introducía sólo un vector y un punto de aplicación antes de cada paso de simulación, y la nueva fuerza resultante era acumulada en una variable situada en **rbsComportamientoLineal** llamada `fuerzaTotal`. Cuando se aplicaba una fuerza también se modificaba el momento de fuerzas (torque) resultante; éste se acumulaba en una variable localizada en **rbsComportamientoRotacional** llamada `momentoFuerzas`. Estas variables eran inicializadas con valor $(0, 0, 0)$. Al hacer un paso de simulación y calcular el siguiente estado del objeto se aplicaban éstas variables acumuladoras, que volvían a tomar valor nulo al final del paso de simulación[1].

Esto resultaba ser muy limitado, ya que sólo nos permitía aplicar fuerzas justo antes de un paso de simulación, esto es, fuerzas **instantáneas**. Por tanto no tendría sentido utilizar el método de Runge-Kutta en el cálculo del siguiente estado, ya que nos daría el mismo resultado que Euler. (Ver Capítulo 2)

Vistas las limitaciones de la primera aproximación decidimos buscar una alternativa que nos permitiese aplicar fuerzas que actúasen en intervalos de tiempo y no únicamente en un paso de simulación. Hemos creado una clase **rbsFuerza**. Cada objeto tiene una estructura en la que se almacenan todas las fuerzas. Dicha estructura es una instancia de la clase **rbsFuerzas**

Cuando sea necesario conocer el valor de la fuerza total que actúa sobre un cuerpo en un instante de tiempo determinado, se consultará a dicha estructura, que a su vez recorrerá y sumará todas las fuerzas almacenadas que estén activas en ese tiempo.

Jerarquía de la clase rbsFuerza

La clase **rbsFuerza** es abstracta, ya que hemos considerado que tanto el punto de aplicación como el vector puedan variar con el tiempo y, por tanto, puedan ser especificadas por el programador en una subclase de ésta. Los métodos que se deberán implementar en los distintos tipos de fuerzas se muestran en el Cuadro 1.4.

[1]Esto es necesario, ya que en otro caso las fuerzas estarían actuando en todos los pasos de simulación.

`rbsFuerza::rbsFuerza(TIPOREAL tiempoInicio)`

Constructor de la clase. Dado que **rbsFuerza** es abstracta, sólo puede ser llamado desde el constructor de una subclase. Se le pasa como parámetro el tiempo de inicio del intervalo en que va a estar activa la fuerza.

`rbsVector3 GetPuntoAplicacion(TIPOREAL tiempo, rbsObjeto* objeto)`

Devuelve el punto de aplicación de la fuerza sobre el objeto en el tiempo indicado.

`rbsVector3 GetVector(TIPOREAL tiempo, rbsObjeto* objeto)`

Devuelve el vector de la fuerza sobre el objeto en el tiempo indicado.

`bool HaFinalizado(TIPOREAL tiempo)`

Devuelve `true` si la fuerza ha finalizado (el tiempo que se pasa como parametro es posterior al final del intervalo) y `false` en caso contrario.(Método abstracto)

`string GetTipo()`

Devuelve el tipo de la fuerza.(Método abstracto)

Cuadro 1.4: Métodos de la clase abstracta **rbsFuerza**

A continuación veremos las distintas subclases de **rbsFuerza** y las relaciones que hay entre ellas. Las fuerzas se caracterizan por un intervalo de tiempo de aplicación y por una función que determina su valor y punto de aplicación. Atendiendo a su duración hemos decidido clasificarlas en:

- **Fuerzas instantáneas**: Se aplican en un único instante de tiempo. Para estas fuerzas sólo es necesario indicar el tiempo inicial del intervalo, ya que el tiempo de finalización coincide con el inicial. Al actuar sólo en un instante de tiempo no es necesario una función que determine la fuerza a lo largo de tiempo, basta con saber el vector fuerza que se quiere aplicar. Por tanto, en este tipo de fuerzas solo necesitamos conocer tiempo de inicio, vector de fuerza y punto de aplicación.

- **Fuerzas temporales**: en este caso es necesario determinar el instante de comienzo y el instante de fin de la fuerza. También será necesario conocer el vector de la fuerza y el punto de aplicación en cada instante del intervalo. Por lo tanto, si queremos definir una nueva fuerza temporal se tendrá que crear una clase que extienda a rbsFuerza-Temporal e implemente los metodos `GetVector()` y `GetPuntoAplicacion()`.

  En nuestro caso hemos implementado una **fuerza temporal invariable**, es decir, que tanto el vector como el punto de aplicación de la fuerza permancen constantes

durante su intervalo de actuación.

- **Fuerzas Perpetuas**: a este tipo pertenecen aquellas fuerzas que tienen un tiempo de inicio, pero no tienen finalización. Al igual que en las fuerzas temporales se precisa conocer el vector y el punto de aplicación de la fuerza en cada instante del intervalo. Por tanto, si se quiere definir una fuerza perpetua habrá que crear una clase que extienda a rbsFuerzaPerpetua e implemente los metodos `GetVector()` y `GetPuntoAplicacion()`.

  Como ejemplo más representativo de fuerza perpetua tenemos el caso de la gravedad. En este caso de fuerza perpetua el punto de aplicación es constante y se corresponde con el centro de masas del objeto. El vector de fuerza varía con el tiempo de la siguiente forma: en primer lugar calculamos la gravedad existente en la altura a la que se encuetra el objeto:

$$g = g_0 \left(1 - 2\frac{objeto \rightarrow GetPosicion() \rightarrow y}{Radio_T}\right)$$

  En donde:

$$g_0 = 9{,}81 m/s^2$$

$$Radio_T = \text{Radio de la Tierra} = 6371000 \text{ m}$$

  La fuerza de gravedad sólo tiene componente y. Por tanto:

$$F_y = g * objeto \rightarrow GetMasa()$$

$$F_{gravedad} = (0, F_y, 0)$$

Gestión de fuerzas dentro de un objeto.

Como ya se ha comentado con anterioridad, cada objeto dispondrá de un vector de fuerzas. En realidad este vector de fuerzas será una estructura implementada por nosotros. La clase encargada de implementar esta estructura es **rbsFuerzas**, que permite las operaciones indicadas en el Cuadro 1.5. Al final de cada paso de simulación, se borrarán todas las fuerzas que ya no actúen sobre el objeto, mediante el método `BorrarFuerzas()`.

1.7.6. Propiedades adicionales

Además de los métodos de **rbsObjeto** antes comentados, se pueden asociar propiedades especificadas por el usuario a un objeto (por ejemplo, su color). Cada propiedad está unívocamente determinada por una cadena que indica su nombre. Pueden añadirse propiedades mediante el método `SetPropiedad()`. Cada propiedad añadida debe ser una subclase de **rbsPropiedad**.

`rbsFuerzas::rbsFuerzas(rbsObjeto* objeto)`

Constructor de la clase. Se le pasa como parámetro el objeto al que va a pertenecer la estructura de fuerzas.

`void AnadirFuerza(rbsFuerza *fuerza);`

Añade la fuerza a la estructura.

`void BorrarFuerzas(TIPOREAL tiempoFinal)`

Elimina aquellas fuerzas que hayan expirado en el instante de tiempo que se pasa como parámetro.

`void BorrarTodasFuerzas()`

Elimina todas las fuerzas de la estructura.

`rbsVector3 GetFuerzaResultante(TIPOREAL tI, TIPOREAL tF)`

Devuelve la fuerza total en el intervalo de tiempo `[tI,tF]` especificado. La fuerza total es la suma de todas las fuerzas que actúan en ese intervalo. Por lo tanto se recorren todas las fuerzas, se comprueban si su intervalo de actuación se solapa con el intervalo que pasamos como parametro y, en caso afirmativo, la fuerza se suma a una variable acumuladora (fuerza total).

`rbsVector3 GetMomentoFuerzasResultante(TIPOREAL tI, TIPOREAL tF)`

Devuelve el momento de fuerzas resultante de todas las fuerzas que actúan en el intervalo indicado. El momento de fuerzas total es la suma de todos los momentos de fuerzas.

Cuadro 1.5: Métodos de la clase **rbsFuerzas**

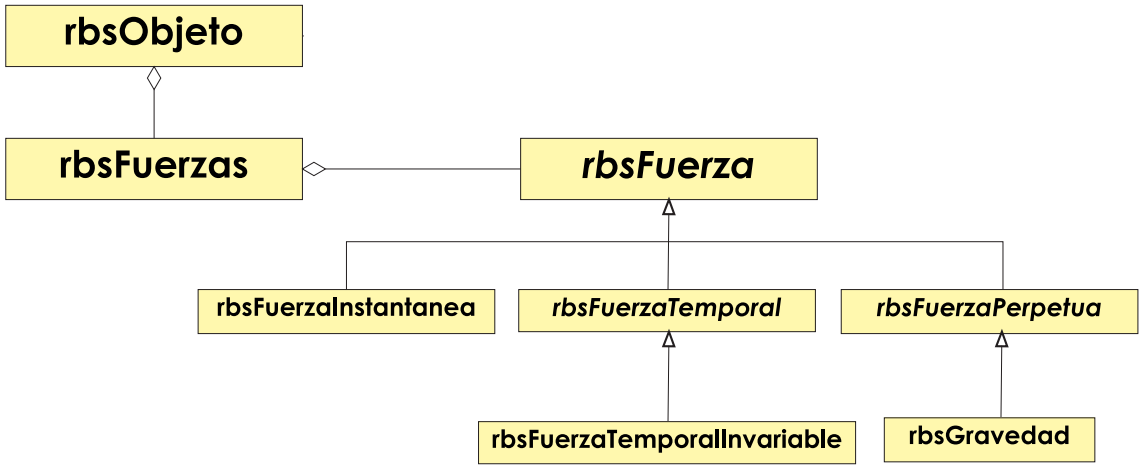


Figura 1.9: Jerarquía de la clase **rbsFuerza**

1.7.7. Bucle principal de la escena

Con las clases implementadas hasta el momento podemos elaborar una primera implementación del *paso de simulación*. Un paso de simulación consiste en el avance del estado de todos los objetos de la escena una determinada cantidad de tiempo. Dicho paso se produce mediante una llamada al método `SiguienteEstado(t)` de la clase **rbsEscena**.

Como aún no tratamos colisiones con otros objetos ni contacto, el paso de simulación resulta ser bastante sencillo, ya que los objetos son totalmente independientes unos de otros y puede realizarse el paso de cada uno por separado. Esto implica que los objetos pueden interpenetrar entre ellos.

rbsEscena :: *SiguienteEstado*(*t*)
para cada objeto *o* de la escena **hacer**
o → SiguienteEstado(t)
fpara
tiempoEscena := *tiempoEscena* + *t*

rbsSolidoRigido :: *SiguienteEstado*(*t*)
si el comportamiento lineal está activo **entonces**
comportamientoLineal→SiguientePosicion(t)
fsi
si el comportamiento rotacional está activo **entonces**
comportamientoRotacional→SiguienteRotacion(t)
fsi

rbsComportamientoLineal :: *SiguientePosicion*(*t*)
$\mathbf{v} := \frac{\mathbf{P}}{MasaObjeto}$
$\mathbf{x}$:= objeto→GetPosicion()
$\langle \mathbf{P'}, \mathbf{x'} \rangle$:= resolutorEDO → CalcularSiguienteEstado($\mathbf{P}$, $\mathbf{x}$, $\mathbf{v}$, $\mathbf{F}(t)$, t)
$\mathbf{P} := \mathbf{P'}$
objeto→SetPosicion($\mathbf{x'}$)

rbsComportamientoRotacional :: *SiguienteRotacion*(*t*)
$\omega := \mathbf{I}^{-1}_{objeto}\mathbf{L}$
$\mathbf{R}$:= objeto→GetRotacion()
$\Omega := \begin{pmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{pmatrix}$
$\langle \mathbf{L'}, \mathbf{R'} \rangle$:= resolutorEDO → CalcularSiguienteEstado($\mathbf{L}$, $\mathbf{R}$, $\Omega\mathbf{R}$, $\tau(t)$,t)
$\mathbf{L} := \mathbf{L'}$
objeto→SetRotacion($\mathbf{R'}$)

Algoritmo 1.3: Primera versión del paso de simulación

Capítulo 2

Resolución de EDOs

Hemos visto anteriormente que en *RBS* la simulación se realiza por pasos. Cada paso consistía en la variación del estado de los objetos de la escena. Dicha variación venía representada por la ecuación 1.13 si se utilizan matrices 3x3 para representar la orientación del objeto. En el caso de que se utilizasen quaterniones, debíamos sustituir la segunda componente de la variación del vector estado por la ecuación 1.18, obteniendo:

$$\frac{d\mathbf{Y}(t)}{dt} = \begin{pmatrix} d\mathbf{x}(t)/dt \\ d\mathbf{q}(t)/dt \\ d\mathbf{P}(t)/dt \\ d\mathbf{L}(t)/dt \end{pmatrix} = \begin{pmatrix} \mathbf{v}(t) \\ \frac{1}{2}\omega\mathbf{q}(t) \\ \mathbf{F}(t) \\ \tau(t) \end{pmatrix} \tag{2.1}$$

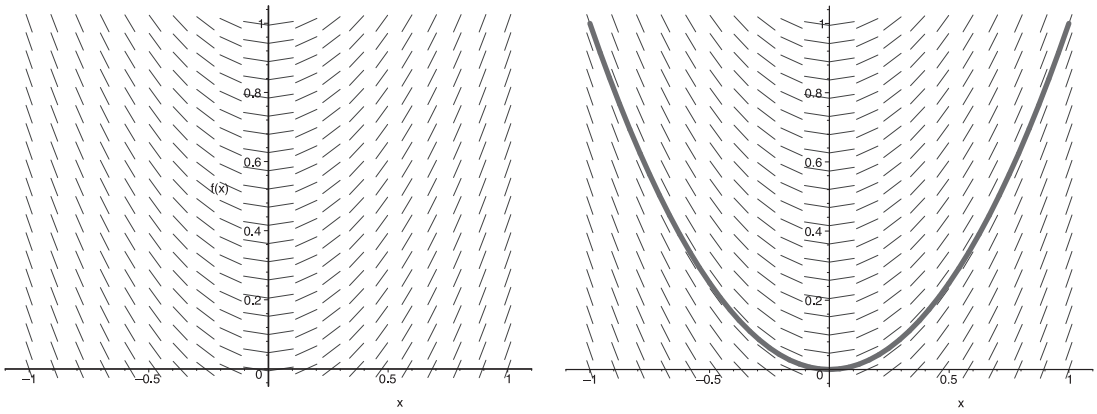
Simplificando el problema al caso de una única dimensión, tratamos de resolver ecuaciones del tipo:

$$\frac{du}{dt} = f(u,t)$$

Es lo que se conoce una *ecuación diferencial ordinaria* (EDO). Gráficamente podemos visualizarla como un campo vectorial (Figura 2.1(a)), donde se indica la variación de u. Particularmente estamos interesados en asociar un *valor inicial* a la ecuación diferencial, de modo que la función que obtengamos como solución de dicha ecuación sea única (Figura 2.1(b)).

Esta clase de problemas son especialmente útiles en la simulación de sólidos rígidos, donde nuestros objetos tienen un **estado inicial**. Partiendo de dicho estado y mediante las cuatro ecuaciones que determinan su variación podemos calcular la evolución de un objeto a lo largo del tiempo.

Dado que la variación de estado de un objeto viene determinada por las fuerzas que actúan sobre él y que dichas fuerzas son especificadas arbitrariamente por el usuario de la



(a) Sin valor inicial (b) Con valor inicial (0,0)

Figura 2.1: Campo vectorial originado por la EDO $du/dt = 2t$

API, no podemos en principio encontrar una solución analítica a las ecuaciones diferenciales. Por esta razón, **la resolución de EDOs se realizará numéricamente**. Dicha resolución consistirá en tomar **pasos discretos a lo largo del tiempo** y evaluar la función f en cada paso.

Hay varios métodos para la resolución numérica de EDOs. En la API *RBS* han sido implementados los métodos de Euler y Runge-Kutta de orden 4. El usuario puede especificar qué método aplicar.

2.1. Método de Euler

Es el método numérico más sencillo. Partiendo de un estado inicial u_0 en el tiempo t_0, podemos realizar una estimación de u en un tiempo $t_0 + h$ (siendo h el tamaño del paso). Dicha estimación viene dada por:

$$u(t_0 + h) = u_0 + h\frac{du}{dt}(t_0) \tag{2.2}$$

En la Figura 2.2 podemos observar una comparación de la solución obtenida por el método de Euler con la función obtenida analíticamente en la ecuación $du/dt = 2t$. El método de Euler con valores de h grandes no es muy preciso. Si disminuimos el tamaño del paso de simulación podemos obtener resultados más exactos. Sin embargo, tamaños de paso demasiado pequeños pueden hacer que la simulación sea poco eficiente. Un buen método de resolución de EDOs debería permitir tamaños de paso mayores.

Para saber de qué modo podemos mejorar este método, tenemos que averiguar previa-

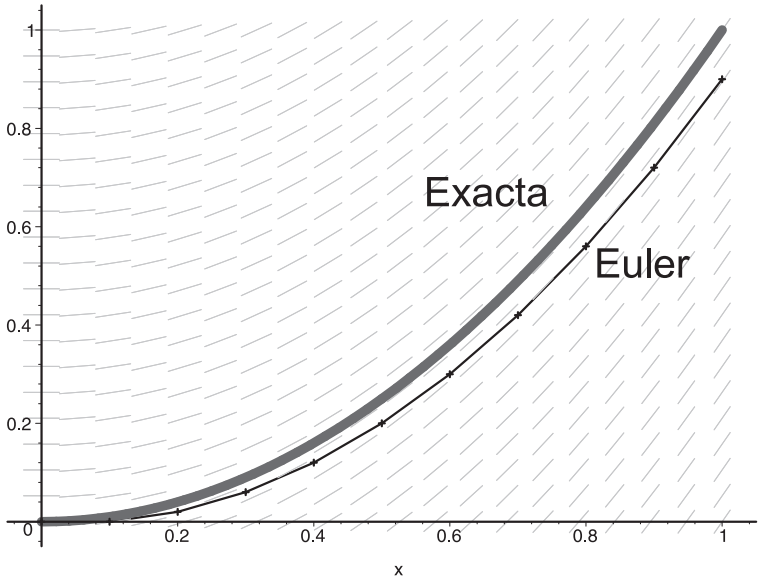


Figura 2.2: Visualización del método de Euler con $h = 0{,}1$

mente el error producido. Suponiendo u contínua, podemos expresar su valor al final del paso de simulación como una serie de Taylor:

$$u(t_0 + h) = u(t_0) + h\frac{du(t_0)}{dt} + \frac{h^2}{2!}\frac{d^2u(t_0)}{dt^2} + \cdots + \frac{h^i}{i!}\frac{d^iu(t_0)}{dt^i} + \cdots \tag{2.3}$$

Si nos quedamos con los dos primeros términos de la serie y descartamos el resto obtenemos la ecuación 2.2. El *error* cometido es igual a la diferencia entre la serie de Taylor completa y el paso de Euler. Vemos que dicho error es del orden del tamaño del paso elevado al cuadrado ($\mathcal{O}(h^2)$). El método de Runge-Kutta de orden 4, que se describe a continuación, permite reducir este error.

2.2. Método de Runge-Kutta de orden 4

Si en la ecuación 2.3 nos quedamos con los primeros cinco términos, en lugar de dos, obtenemos:

$$u(t_0 + h) = u(t_0) + h\frac{du(t_0)}{dt} + \frac{h^2}{2!}\frac{d^2u(t_0)}{dt^2} + \frac{h^3}{3!}\frac{d^3u(t_0)}{dt^3} + \frac{h^4}{4!}\frac{d^4u(t_0)}{dt^4} \tag{2.4}$$

El problema pasa ahora por evaluar las derivadas segunda, tercera y cuarta de u. En [BF98] se muestra una explicación del cálculo completo.

La fórmula para el cálculo de $u(t_0 + h)$ resulta:

$$u(t_0 + h) = u_0 + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4$$

donde:

$$\begin{aligned} k_1 &= hf(u_0, t_0) \\ k_2 &= hf\left(u_0 + \frac{k_1}{2}, t_0 + \frac{h}{2}\right) \\ k_3 &= hf\left(u_0 + \frac{k_2}{2}, t_0 + \frac{h}{2}\right) \\ k_4 &= hf\left(u_0 + \frac{k_3}{2}, t_0 + h\right) \end{aligned}$$

Como hemos eliminado los términos de la serie de Taylor a partir del sexto, el error cometido es de $\mathcal{O}(h^5)$. Esta mejora[1] en la precisión se produce a costa de evaluar cuatro veces la función f. No obstante, al permitir tamaños de paso más grandes, este método supera al de Euler en eficiencia. Por tanto, **este será el método utilizado por defecto en *RBS***.

2.3. Detalles de implementación

Como ya hemos visto, es necesario disponer de una función que resuelva el problema del cálculo del siguiente estado aplicando uno de los dos métodos vistos. Para ello crearemos una clase **rbsEDO** que nos proporcionará dicha función. Según el método de integración los cálculos que se realizan dentro de la función cambian, por tanto la clase **rbsEDO** será abstracta y será necesario crear una subclase que la implemente para cada uno de los métodos. En nuestro caso tenemos dos métodos (Euler y Runge-Kutta), por tanto tendremos dos clases que implementarán a **rbsEDO**. En la figura 2.3 podemos ver el diagrama de clases empleado.

Por tanto la principal función de **rbsEDO** será la encargada de calcular el siguiente estado aplicando Euler o Runge-Kutta. La función está definida de la siguiente forma:

```
virtual void Resolver (TIPOREAL estadoInicio[],TIPOREAL estadoFinal[],
                       int longitudVectorEstado,TIPOREAL tiempoInicio,
                       TIPOREAL tiempoFin, rbsObjeto* objeto,
                       dydtfunc dydt)=0
```

[1] En realidad sólo tenemos una mejora para tamaños de paso h comprendidos entre 0 y 1. En ese caso, $h^2 > h^5$. Valores entre 0,01 y 0,03 suelen ser buenas medidas para el tamaño del paso.

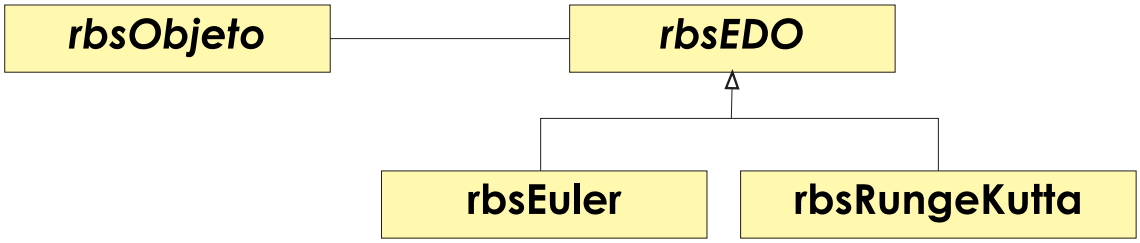


Figura 2.3: Diagrama de herencias para la clase **rbsEDO**

Como vemos, se necesita conocer el estado de partida, el tiempo actual y el tiempo final (que es igual a tiempo actual + tiempo de paso) y la función que calcula la derivada del estado (`dydt`).

Este método será implementado por las distintas clases que hereden de **rbsEDO** aplicando el método de integración oportuno.

Con respecto a la función `dydt` que se pasa como parámetro en `Resolver` y calcula la derivada, esta función tiene la siguiente cabecera:

```
typedef void (*dydtfunc)(TIPOREAL tiempo, TIPOREAL tiempoPaso,
                         TIPOREAL y[], TIPOREAL ydot[],
                         rbsObjeto* objeto);
```

donde `y` es el estado actual, `ydot` la derivada del estado actual, `tiempo` es el tiempo de simulación en que nos encontramos y `tiempoPaso` es el paso de simulación que se va a dar (es necesario conocerlo para poder determinar las fuerzas que actúan).

El sistema se puede dividir en comportamiento rotacional y comportamiento lineal. Vamos a hacer los cálculos de forma independiente, pues ya hemos visto que dichos comportamientos pueden estar activos o no. Por lo tanto, se necesita definir dos funciones, que se corresponden con la derivada del comportamiento lineal y derivada del comportamiento rotacional.

El uso del resolutor se hace en `SiguientePosicion(TIPOREAL tiempo)` de **rbsComportamientoLineal** y en `SiguienteRotacion(TIPOREAL tiempo)` de la clase **rbsComportamientoRotacional**. Además en estas clases están definidas las funciones `DerivadaLineal` y `DerivadaRotacional` que tendrán la cabecera vista arriba y serán las funciones dinámicas que pasaremos al método `Resolver` del resolutor.

El usuario puede elegir el método de resolución que desea emplear en el cálculo del estado siguiente. Cada objeto tendrá su forma de obtener el siguiente estado y podrá ser seleccionada a través de las siguientes funciones:

`void rbsObjeto::SetMetodoDerivada(rbsMetodoDerivada nuevoMetodo)`

Elige el método de integración que será aplicado. `rbsMetodoDerivada` es un enumerado en donde se guardan los tipos disponibles: `RBSEULER` o `RBSRUNGEKUTTA`.

`rbsMetodoDerivada rbsObjeto::GetMetodoDerivada()`

Devuelve el tipo método de integración activo.

Capítulo 3

Detección de colisiones

3.1. Introducción

En este capítulo veremos cómo detectar las colisiones entre los distintos tipos de geometrías de las que se dispone en la API *RBS*(ver Figura 1.7). A la hora de detectar las colisiones y calcular los puntos de contacto que se han producido, nos encontramos con varias aproximaciones. Por ejemplo, podemos encerrar los sólidos rígidos en cilindros, esferas, paralelepípedos, etc. y calcular las colisiones entre estas geometrías básicas. No obstante, estas aproximaciones no nos dan, en general, los puntos de contacto reales que se han producido en la colisión, sino puntos aproximados. Por otro lado, dado que estas geometrías básicas ocupan más volúmen que los sólidos reales a los que "envuelven" podrían detectarse colisiones entre objetos que no están ocurriendo realmente (Figura 3.1). Por lo tanto, este enfoque no es adecuado para nuestros propósitos de crear un API que se aproxime, en la medida de lo posible, a la realidad.

Así pues, en *RBS* **se implementan diversos algoritmos específicos para cada geometría** que determinen de modo exacto los puntos de contacto. No obstante, también utilizaremos cajas envolventes para descartar colisiones entre objetos (esto se verá en la Sección 5.1). En las siguientes secciones iremos viendo los algoritmos empleados y examinaremos su eficiencia.

A partir de ahora, para todos nuestros algoritmos debemos tener en cuenta que por errores de redondeo en las operaciones necesitamos introducir una constante TOL, cuyo valor será no mayor de 0.01. Los valores por debajo de dicha constante se considerarán 0. Del mismo modo, los valores cuya diferencia sea menor que TOL se considerarán iguales.

Todos los algoritmos **devolverán un vector de contactos**. Cada contacto será una instancia de la clase **rbsContacto**, que modela un punto de contacto entre dos objetos.

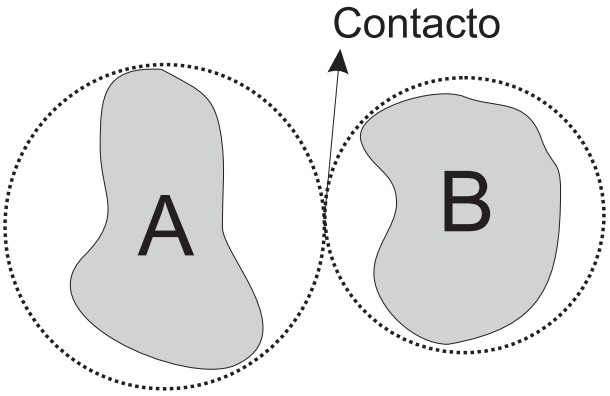


Figura 3.1: Contacto detectado entre dos cuerpos con geometría arbitraria (gris), utilizando esferas envolventes. Se ha detectado una colisión, aunque los cuerpos no están en contacto.

3.2. La clase rbsContacto

En todos los algoritmos sólo encontramos dos tipos de contactos:

- Contactos entre vértice/cara.
- Contactos entre arista/arista.

Los contactos entre vértice/vértice, y vértice/arista se consideran casos degenerados y no se detectan ni se tratan en la simulación.

Los atributos que componen esta clase son:

- Las referencias a los dos **objetos** que están en contacto. Estos objetos serán referidos como A y B.
- El **punto de contacto**. Se especifica en coordenadas globales de la escena.
- **Vector normal** de contacto. Por convenio, este vector debe apuntar hacia el objeto A
- **Arista 1** de contacto. Pertenece al primer objeto y sólo se aplica en contactos arista/arista.
- **Arista 2** de contacto. Pertenece al segundo objeto y sólo se aplica en contactos arista/arista.
- Un valor booleano que indica si se trata de un contacto entre **vértice/cara** o entre **arista/arista**.

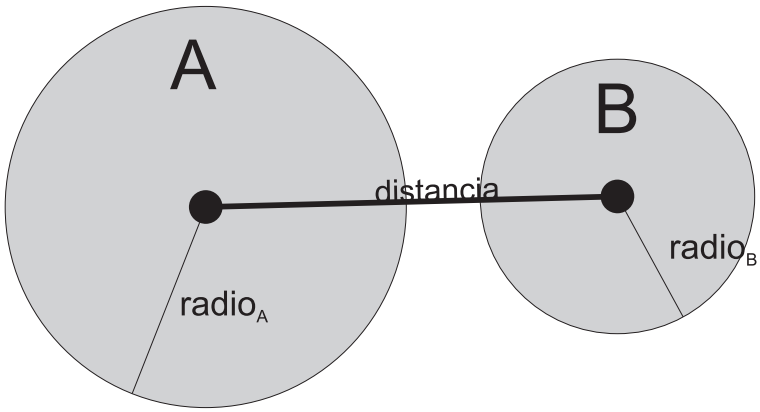


Figura 3.2: Colision entre esferas. Como la distancia entre los centros es mayor que la suma de los radios, las esferas no están colisionando.

3.3. Colisión entre esferas

Éste es el algoritmo de colisón más sencillo. Sólo se detectará como máximo un punto de contacto, que se considerará vértice/cara. Para ello, una primera idea sería calcular la distancia que hay entre los centros de las esferas y compararla con la suma de los radios (Figura 3.2). No obstante, al calcular la distancia entre dos puntos (los centros), es necesario utilizar una raíz cuadrada, que computacionalmente es bastante costosa de realizar. Para mejorar la eficiencia del algoritmo, eliminaremos esta raíz.

Dadas las posiciones de las esferas X_a y X_b y sus respectivos radios r_a y r_b, la intersección existe si se cumple que:

$$|X_a - X_b| \leq r_a + r_b$$

Elevando al cuadrado, esto equivale a:

$$(X_{ax} - X_{bx})^2 + (X_{ay} - X_{by})^2 + (X_{az} - X_{bz})^2 \leq (r_a + r_b)^2$$

Con lo que hemos eliminado la raíz cuadrada. Si llamamos D a la parte izquierda de la ecuación, es decir, $D = (X_{ax} - X_{bx})^2 + (X_{ay} - X_{by})^2 + (X_{az} - X_{bz})^2$ y R a la parte derecha, $R = (r_a + r_b)^2$, concluímos que:

- Si $D = R$ (con tolerancia TOL), entonces se produce colisión entre ambas esferas. El punto de contacto estará en la recta que une los centros y es muy sencillo deducirlo a partir de los radios.
- Si $D < R$, entonces se produce interpenetración entre las esferas.
- Si $D > R$, entonces no se produce ni colisión ni interpenetración.

Por lo tanto, el coste de este algoritmo es constante ($\mathcal{O}(1)$).

3.4. Colisión entre esfera y paralelepípedo

La idea básica de este algoritmo se basa en que la esfera podrá colisionar con una **arista** o con **una de las caras** del paralelepípedo. Por lo tanto, trataremos la detección de colisión esfera-arista y esfera-cara. Por otra parte, debemos tener en cuenta que en una hipotética colisión entre ambos cuerpos **sólo se producirá un punto de contacto**.

El paralelepípedo viene definido por su posición y orientación y por el alto, ancho y largo que ocupa. Por lo tanto, antes de nada necesitaremos generar las aristas y caras del paralelepipedo, teniendo en cuenta sus características y su rotación. Las aristas vienen definidas por dos puntos (sus extremos) y las caras (planos) por dos vectores directores y el punto origen de ambos.

Puesto que los planos del paralelepípedo incluyen las aristas, deberemos comenzar primero a calcular la colisión esfera-arista, porque en caso contrario nunca se detectaría la colisión con una arista. En efecto, imaginemos que la esfera colisiona contra una arista; si primero comprobamos la intersección con los planos del paralelepípedo, detectaría la colisión con el plano en el que está la arista y el algoritmo devolvería el contacto relativo a una colisión con un plano, por lo que la normal del contacto no sería la correcta. Por lo tanto, debemos comprobar primero las colisiones entre esfera-arista y si no se ha producido colisión con ninguna arista, entonces comprobar si se produce colisión con algúna cara. Por otra parte, si se ha producido una colisión con alguna arista, entonces ya no comprobaremos la colisión esfera-cara.

En cualquier caso, el punto de contacto (si lo hubiese) se considera de tipo vértice/cara.

3.4.1. Colisión esfera-arista

La esfera sólo podrá colisionar con una única arista del paralelepípedo, por lo que en cuanto se detecte una colisión, el algoritmo terminará. En primer lugar determinaremos la distancia del centro de la esfera a la arista actual. Para ello aplicamos la fórmula para hallar la distancia de un punto a la recta:

$$\textbf{distancia(P,r)} = \frac{|\overline{AP} \times v|}{|v|}$$

donde:

- **P** es el punto del que queremos hallar la distancia (en nuestro caso el centro de la circunferencia).
- **r** es la recta que contiene a la arista.

- A es un punto auxiliar de la recta.
- v es el vector director de la recta.

Sea r_e el radio de la esfera y d la distancia calculada. Se nos presentará uno de los siguientes casos:

- Si $d = r_e$, con tolerancia TOL, entonces se produce colisión entre la esfera y la recta que contiene a la arista. Puesto que la recta es infinita, para que realmente se produzca una colisión entre esfera y paralelepípedo, el punto de contacto ha de pertenecer al paralelepípedo. Para ello tenemos un método dentro de la geometría del paralelepípedo que nos indica si un punto pertenece o no a él. Por lo tanto, antes de esta comprobación necesitaremos conocer el hipotético punto de contacto. Dicho punto estará en la intersección entre la recta que contiene a la arista y la recta perpendicular a la anterior y que pasa por el centro de la esfera. El punto obtenido se encuentra a una distancia d (en este caso r_e) del centro de la esfera.
- Si $d < r_e$ y el punto de contacto calculado anteriormente está dentro del paralelepípedo, entonces se produce interpenetración entre ambos sólidos.
- En otro caso, no se produce ni colisión ni interpenetración.

En el peor de los casos necesitamos recorrer las 12 aristas que tiene un paralelepípedo, por lo tanto, el coste de este algoritmo es constante.

3.4.2. Colisión esfera-cara

A esta parte del algoritmo llegamos si no ha habido colisión ni interpenetración con ninguna de las 12 aristas del paralelepípedo. La esfera sólo podrá colisionar con una única cara del paralelepípedo, por lo que en cuanto se detecte una colisión, el algoritmo terminará. En primer lugar necesitamos determinar la distancia del centro de la esfera a la cara actual. Para ello, aplicamos la fórmula para hallar la distancia de un punto un plano:

$$\textbf{distancia(P,}\Pi\textbf{)} = \frac{|\overline{AP} \cdot n|}{|n|}$$

en donde,

- P es el punto del que queremos hallar la distancia al plano (en nuestro caso el centro de la circunferencia).
- Π es el plano (el plano que contiene a la cara del paralelepípedo).

- A es un punto cualquiera del plano Π.
- n es la normal del plano (calculada como el producto vectorial de los vectores directores que definen al plano).

Sea el radio de la esfera r_e y d la distancia calculada. Se nos presentarán los siguientes casos:

- Si $d = r_e$, con tolerancia TOL, entonces se produce colisión entre la esfera y el plano que contiene a la cara. Puesto que el plano es infinito, para que realmente se produzca una colisión entre esfera y paralelepípedo el punto de contacto ha de pertenecer a este último. Para ello aplicamos de nuevo el método que hay dentro de la geometría del paralelepípedo que nos indica si un punto pertenece o no a él. Por lo tanto, necesitaremos conocer de nuevo el hipotético punto de contacto, el cual estará en la recta cuyo vector director es la normal del plano y a una distancia d (en este caso r_e) del centro de la esfera.
- Si $d < r_e$ y el punto de contacto calculado anteriormente está dentro del paralelepípedo, entonces se produce interpenetración entre ambos sólidos.
- En otro caso, no se produce ni colisión ni interpenetración.

En el peor de los casos necesitamos recorrer las 6 caras que tiene un paralelepípedo, por lo tanto, el coste de este algoritmo es constante.

Por lo tanto, el coste del algoritmo completo (colisión entre esfera/arista + colisión entre esfera/cara) es constante.

3.5. Colisión entre paralelepípedos

En este apartado se va a tratar el cálculo de los puntos de contacto existentes en la colisión de dos paralelepípedos. Puede haber más de un punto.

Los contactos pueden ser de dos tipos:

- Vértice contra una cara.
- Arista contra arista.

Antes de ver si hay puntos de contacto comprobaremos que los paralelepípedos no están separados ni interpenetran. Para ello utilizaremos el algoritmo que describiremos en la siguiente sección.

3.5.1. Comprobación de existencia de contacto

Vamos a ver cómo averiguar si dos paralelepipedos están en contacto, interpenetran o están separados. Para ello haremos dos comprobaciones simétricas que se corresponden con la existencia de solapamiento entre las proyecciones de los paralelepípedos en el sistema local de uno de ellos.

Partimos de dos paralelepípedos A y B. Vamos a comprobar primero si las proyecciones de A y B en el sistema de coordenadas de A se solapan. Posteriormente tendríamos que realizar la misma comprobación, pero en el sistema de coordenadas de B.

Primero se transforman los vertices de B al sistema de coordenadas de A. De estos vertices calculamos sus límites:

$$(maximoX_B, maximoY_B, maximoZ_B, minimoX_B, minimoY_B, minimoZ_B)$$

Calculamos también los limites del paralelepipedo A:

$$maximoX_A = \frac{Ancho_A}{2}$$
$$maximoY_A = \frac{Alto_A}{2}$$
$$maximoZ_A = \frac{Largo_A}{2}$$
$$minimoX_A = -\frac{Ancho_A}{2}$$
$$minimoY_A = -\frac{Alto_A}{2}$$
$$minimoZ_A = -\frac{LargoA}{2}$$

Las proyecciones serían los siguientes segmentos, para cualquier objeto O:

- Proyección en eje X: $[minimoX_O, maximoX_O]$
- Proyección en eje Y: $[minimoY_O, maximoY_O]$
- Proyección en eje Z: $[minimoZ_O, maximoZ_O]$

A continuación comprobamos si se solapan cada par de proyecciones:

- $[minimoX_A, maximoX_A]$ con $[minimoX_B, maximoX_B]$

- $[minimoY_A, maximoY_A]$ con $[minimoY_B, maximoY_B]$
- $[minimoZ_A, maximoZ_A]$ con $[minimoZ_B, maximoZ_B]$

A la hora de comprobar si un segmento se solapa con otro hay tres posibilidades: interpenetran, están separados o en contacto. Interpenetran cuando tienen más de un punto en común, están separados si no tienen ninguno y en contacto cuando sólo tienen uno.

Si los tres interpenetran, devolveremos que las proyecciones (en el caso del sistema de coordenadas local a A) interpenetran. Si una de las parejas de proyecciones no solapa, devolveremos que las proyecciones estan separadas. En cualquier otro caso, devolveremos que las proyecciones se encuentran en contacto.

Realizamos los mismos cálculos pero en coordenadas locales a B.

Con la información obtenida por estas dos comprobaciones podremos conocer si los paralelepipedos interpenetran, estan separados o estan en contacto: Interpenetrarán si las dos comprobaciones nos dieron que interpenetraban, estarán separados si una de ellas devolvió que estaban separados y estarán en contacto en cualquier otro caso. Este proceso se encuentra detallado en el Algoritmo 3.1.

Una vez que sabemos que están en contacto tenemos que calcular los posibles puntos de contacto. Esto es lo que veremos a continuación.

3.5.2. Colisión Vértice-Cara

Tendremos que comprobar la colisión de los vértices del paralelepípedo B con las caras del paralelepípedo A y viceversa. Para ello, trabajaremos con las coordenadas locales de A.

En primer lugar transformaremos los vertices de B al sistema de coordenadas local de A. Una vez tengamos los vértices en coordenadas locales, recorreremos todos los vértices y miraremos si alguno de ellos colisiona con alguna de las caras de A. La comprobación es muy sencilla, basta con comprobar que cada vértice está en los limites de la cara y se encuentra en el mismo plano que ésta. Por ejemplo, para comprobar si un vértice con coordenadas $V(v_x, v_y, v_z)$ está en contacto con la cara superior del paralelepipedo se hará la siguiente comprobación:

- $v_y = (Alto_A)/2$
- $(-Ancho_A/2) < v_x < (Ancho_A/2)$
- $(-Largo_A/2) < v_z < (Largo_A/2)$

Estado rbsAlgoritmoColisionParalelepipedos :: $Interpenetran()$
    $estado1 := SolapanProyecciones(A, B)$
    $estado2 := SolapanProyecciones(B, A)$
    si (estado1 = `INTERPENETRAN`) $\wedge$ (estado2 = `INTERPENETRAN`) **entonces**
        devolver `INTERPENETRAN`
    sino si (estado1 = `SEPARADOS`) $\vee$ (estado2 = `SEPARADOS`) **entonces**
        devolver `SEPARADOS`
    sino
        devolver `CONTACTO`
    fsi

Estado rbsAlgoritmoColisionParalelepipedos :: $SolapanProyecciones(A, B)$
    $verticesB := CalcularVerticesGlobales(B)$
    para cada vértice v de $verticesB$ **hacer**
        transformar v a coordenadas de A
    fpara
    $\langle maxXB, minXB, maxYB, minYB, maxZB, minZB\rangle := CalcularLimites(verticesB)$
    $maxXA := A.ancho/2$
    $minXA := -A.ancho/2$
    $maxYA := A.alto/2$
    $minYA := -A.alto/2$
    $maxZA := A.largo/2$
    $minZA := -A.largo/2$
    $estado1 := SolapanSegmentos(minXA, maxXA, minXB, maxXB)$
    $estado2 := SolapanSegmentos(minYA, maxYA, minYB, maxYB)$
    $estado3 := SolapanSegmentos(minZA, maxZA, minZB, maxZB)$
    si (estado1 = `INTERPENETRAN`) $\wedge$ (estado2 = `INTERPENETRAN`)
            $\wedge$ (estado3 = `INTERPENETRAN`) **entonces**
        devolver `INTERPENETRAN`
    sino si (estado1=`SEPARADOS`) $\vee$ (estado2 = `SEPARADOS`)
            $\vee$ (estado3 = `SEPARADOS`) **entonces**
        return `SEPARADOS`
    sino
        return `CONTACTO`
    fsi

Algoritmo 3.1: Comprobación de estado entre dos paralelepipedos.

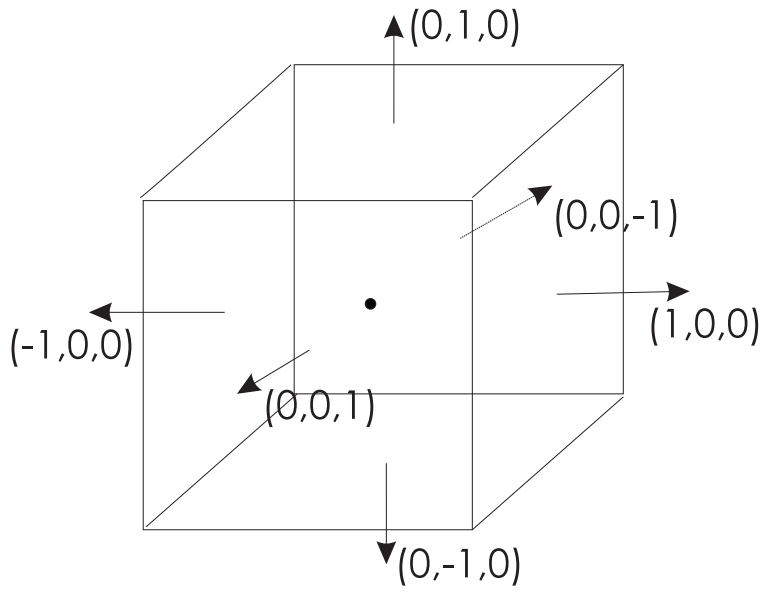


Figura 3.3: Normales de un paralelepípedo en su sistema de coordenadas local

Los casos $v_x = \pm Ancho_A/2$ y $v_z = \pm Largo_A/2$ son tratados en colisión arista-arista. Tendríamos que hacer 6 comprobaciones de este tipo por cada vértice, aunque en caso de encontrar un contacto no es necesario seguir comprobando el resto de caras.

La normal de la colisión resulta fácil de calcular en coordenadas locales: sólo habría que considerar la normal de la cara con la que se colisione (Figura 3.3).

Por último tendremos que transformar el punto de contacto y la normal obtenidos al sistema de coordenadas globales.

3.5.3. Colisión Arista-Arista

En este caso trabajaremos en coordenadas globales. Para cada arista del paralelepípedo A comprobaremos si colisiona con alguna del paralelepípedo B. Para ello:

- Comprobamos si las rectas que contienen a las aristas se cortan. Sean $\mathbf{u}$ y $\mathbf{v}$ los vectores directores de dichas rectas. Si las rectas no son paralelas ($|\mathbf{u} \times \mathbf{v}| \neq 0$) tendremos que determinar la distancia que existe entre las rectas en su punto más próximo.

$$distancia = \frac{\left|\overline{PQ} * (\mathbf{u} \times \mathbf{v})\right|}{|\mathbf{v}|}$$

  donde P es un punto de la recta que tiene como vector $\mathbf{v}$ y Q un punto de la recta que tiene como vector $\mathbf{u}$

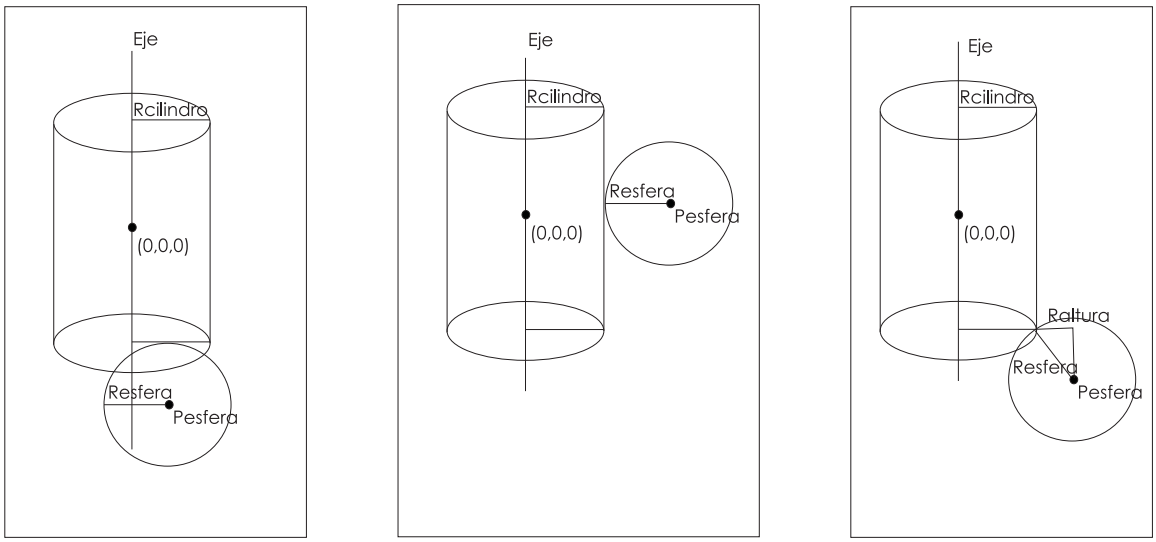


Figura 3.4: Posibilidades de contacto entre esfera y cilindro.

Si la distancia es menor que un cierto umbral TOL, las rectas se cortan.

- Una vez que detectamos que las rectas se cortan tenemos que buscar el punto de corte y comprobar que dicho punto esté comprendido en las aristas. La normal del contacto será el producto vectorial de los vectores $\mathbf{u}$ y $\mathbf{v}$. Como por convenio la normal debe estar dirigida hacia A, debemos comprobar que esto es realmente así; en caso contrario, debemos multiplicarla por -1.

3.6. Colisión entre esfera y cilindro

En este caso, y al igual que sucede con el resto de algoritmos de colisión en los que uno de los objetos involucrados es una esfera, se devuelve un único punto de contacto. Hay tres posibilidades de contacto entre una esfera y un cilindro (Figura 3.4):

- Esfera contra una de las bases del cilindro.
- Esfera contra el lateral del cilindro.
- Esfera contra una de las circunferencias que delimitan cada base.

Cualquiera de estas tres posibilidades dará como resultado un contacto vértice/cara.

Para hacer las comprobaciones pasamos la esfera a coordenadas locales del cilindro; de este modo los cálculos serán más sencillos. Una vez detectado el punto de contacto deberemos transformarlo a coordenadas globales.

3.6.1. Colisión esfera-base

Para que se produca una colisión esfera-base, la distancia del centro de la esfera (P_{Esfera}) al eje del cilindro tiene que ser menor o igual que el radio del cilindro:

$$Radio^2_{Cilidro} \geq (P_{Esfera}(x)^2 + P_{Esfera}(z)^2)$$

En caso de no cumplirse tendríamos que ver qué ocurre con las otras dos posibilidades de colisión esfera-cilindro. Si se cumple esta condicion hay que comprobar que la esfera se sitúe en la altura correspondiente a este tipo de colisión. Hay dos posibilidades:

- Esfera-base superior:

$$P_{Esfera}(y) = \frac{Alto_{Cilidro}}{2} + Radio_{Esfera}$$

- Esfera-base inferior:

$$P_{Esfera}(y) = \frac{-Alto_{Cilidro}}{2} - Radio_{Esfera}$$

Si se cumple una de estas condiciones tendremos que el punto de contacto es:

$$puntoContacto = \left(P_{Esfera}(x), \frac{\pm Alto_{Cilidro}}{2}, P_{Esfera}(z)\right)$$

En el caso de que no se cumpliera ninguna de esas dos condiciones puede ocurrir lo sigiente:

- Que haya penetración:

$$\frac{-Alto_{Cilidro}}{2} - Radio_{Esfera} < P_{Esfera}(y) < \frac{Alto_{Cilidro}}{2} + Radio_{Esfera}$$

- Que estén separados:

$$\left(\frac{-Alto_{Cilidro}}{2} - Radio_{Esfera} > P_{Esfera}(y)\right) \vee \left(\frac{Alto_{Cilidro}}{2} + Radio_{Esfera} < P_{Esfera}(y)\right)$$

3.6.2. Colisión esfera-lateral

En este caso se tienen que dar dos condiciones para que haya contacto:

- El centro de la esfera debe estar comprendido entre la base superior e inferior:

$$\frac{-Alto_{Cilindro}}{2} \leq P_{Esfera}(y) \leq \frac{Alto_{Cilidro}}{2}$$

- La distancia de la esfera al eje del cilindro debe ser igual (con tolerancia TOL) a la suma del radio del cilindro y el de la esfera:

$$P_{Esfera}(x)^2 + P_{Esfera}(z)^2 = (Radio_{Esfera} + Radio_{Cilindro})^2$$

Si se cumple sólo la primera condición, podemos estar en el caso de que esta última distancia sea mayor y por tanto, los objetos estén separados o que la distancia sea menor y en este caso habrá penetración.

El punto de contacto en caso de cumplirse los requisitos anteriores sería:

$$puntoConctacto = P_{Esfera} + direccion * Radio_{Esfera}$$

donde *direccion* es el vector normalizado que se dirige desde el centro de la esfera al punto del eje del cilindro con la misma altura.

$$direccion = Normalizar((0, P_{Esfera}(y), 0) - P_{Esfera})$$

3.6.3. Colisión esfera-circunferencia

Este es el caso más complicado. Ocurre cuando se cumplen las siguientes condiciones:

- El centro de la esfera debe estar en uno de los siguientes intervalos de altura:
  - En el caso de colisionar con la circunferencia superior:

$$\frac{Alto_{Cilindro}}{2} + Radio_{Esfera} > P_{Esfera}(y) > \frac{Alto_{Cilindro}}{2}$$

  - En el caso de colisionar con la circunferencia inferior:

$$-\frac{Alto_{Cilindro}}{2} - Radio_{Esfera} > P_{Esfera}(y) > -\frac{Alto_{Cilindro}}{2}$$

- La distancia entre el centro de la esfera y el eje del cilindro debe ser mayor o igual que el radio de la esfera y menor o igual que la suma de los radios:

$$Radio^2_{Esfera} \leq P_{Esfera}(x)^2 + P_{Esfera}(z)^2 \leq (Radio_{Esfera} + Radio_{Cilindro})^2$$

Una vez que hayamos visto que se cumplen tendremos que hallar el punto de contacto. Para ello calcularemos el radio de la esfera en la altura $\pm Alto_{Cilindro}$ (+ ó − en función de si es contacto con la circunferencia de abajo o con la de arriba). Para calcular este radio (que llamaremos $Radio_{altura}$) aplicamos el Teorema de Pitágoras:

$$hipotenusa^2 = catetoA^2 + catetoB^2$$

$$Radio^2_{Esfera} = Radio^2_{Altura} + \left(|P_{Esfera}(y)| - \left|\frac{Alto_{Cilindro}}{2}\right|\right)^2$$

Despejando $Radio_{Altura}$:

$$Radio_{Altura} = \sqrt{Radio^2_{Esfera} - \left(|P_{Esfera}(y)| - \left|\frac{Alto_{Cilindro}}{2}\right|\right)^2}$$

Una vez que tenemos este valor, podemos calcular el punto de contacto del mismo modo que en la sección 3.6.2.

$$puntoContacto = (P_{Esfera}(x), \pm\frac{Alto_{Cilindro}}{2}, P_{Esfera}(z)) + direccion * Radio_{Altura}$$

donde:

$$direccion = Normalizar((0, \pm\frac{Alto_{Cilindro}}{2}, 0) - (P_{Esfera}(x), \pm\frac{Alto_{Cilindro}}{2}, P_{Esfera}(z)))$$

3.7. Colisión entre mallas convexas

El problema de detección de colisión entre mallas que describen una geometría arbitraria es el más complejo de los aquí comentados. El algoritmo descrito en esta sección sólo se aplica a mallas que definen un **poliedro convexo**. Un poliedro es convexo si tomados dos puntos cualquiera de su interior, determinan un segmento de recta que también es interior (Figura 3.5).

La técnica presentada aquí se basa en el hecho de que **dos poliedros convexos no interpenetran sí y sólo sí existe un *plano separador* entre ellos**. Un plano separador entre dos poliedros A y B es un plano tal que los poliedros se sitúan en los lados opuestos de

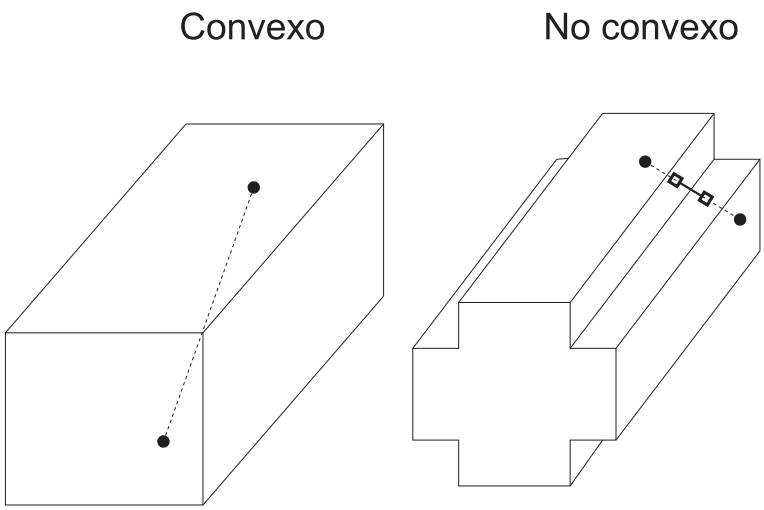


Figura 3.5: Poliedros convexo y no convexo

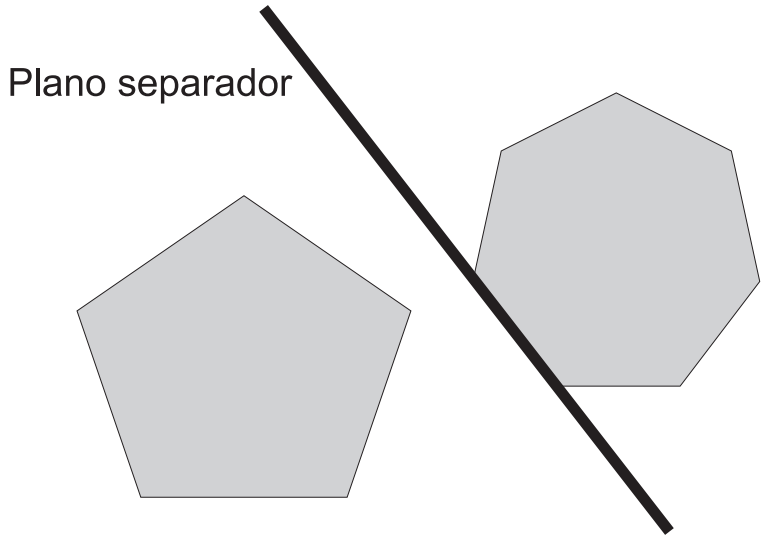


Figura 3.6: Idea del plano separador. Por motivos de claridad mostramos una proyección ortogonal.

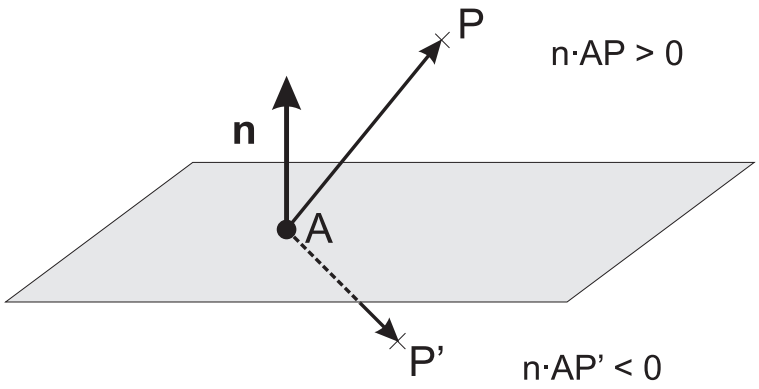


Figura 3.7: Cálculo del semiespacio al que pertenece un punto.

éste (Figura 3.6). Entendemos por "lados" de un plano a cada uno de los dos semiespacios generados por dicho plano. Para averiguar a qué semiespacio pertenece un punto P, basta con realizar el producto escalar entre un vector **n** normal al plano y un vector que una un punto A del plano con P (Figura 3.7).

Así pues, para comprobar si un plano es separador entre dos objetos A y B, basta con comprobar que todos los vértices de A quedan a un lado del plano y todos los vértices de B quedan en el lado opuesto.

Ahora bien, el problema consiste ahora en la búsqueda del plano separador. Para ello es necesario recorrer todas las caras de cada poliedro, formar un plano con cada una y comprobar si es separador. Si ningún plano formado de este modo es separador, combinamos cada arista de un poliedro con cada arista del otro. Cada par de aristas formará un plano que puede ser separador. Si no encontramos ningún plano separador, los objetos están interpenetrando. Este proceso se describe en el Algoritmo 3.2.

Esta búsqueda resulta terriblemente costosa desde el punto de vista de la eficiencia. Sin embargo, una de las ventajas de esta técnica es que **en la mayoría de los casos el plano separador sigue separando los objetos tras el siguiente paso**, ya que la distancia que avanzan dichos objetos es muy corta.

La búsqueda del plano separador en la primera llamada al algoritmo es inevitable. Cuando los objetos dejan de estar a ambos lados de un plano separador (Figura 3.8), se debe buscar otro. Para ello podemos examinar las aristas y caras adyacentes a aquellas que formaron el plano anterior. Sin embargo, esto requeriría una estructura especial para almacenar las mallas, de modo que se tuviese un acceso eficiente a las caras/aristas adyacentes a una dada. Además la búsqueda de un plano separador resulta en general tan poco frecuente que podemos realizarla cada vez desde cero.

Otra ventaja de esta técnica de detección de colisiones es que todos los contactos, si los hay, se encuentran en el plano separador, por lo que sólo tenemos que comprobar las

```
proc BuscarPlanoSeparador(A, B)
    para cada cara F_A de A hacer
        Formar un plano Π con F_A
        si Π es separador entonces
            devolver Π
        fsi
    fpara

    para cada cara F_B de B hacer
        Formar un plano Π con F_B
        si Π es separador entonces
            devolver Π
        fsi
    fpara

    para cada arista e_A de A hacer
        para cada arista e_B de B hacer
            P := punto de inicio de e_A
            n := e_A × e_B
            Formar un plano Π que pase por P y tenga normal n
            si Π es separador entonces
                devolver Π
            fsi
        fpara
    fpara

    devolver NULL
fproc
```

Algoritmo 3.2: Búsqueda de un plano separador

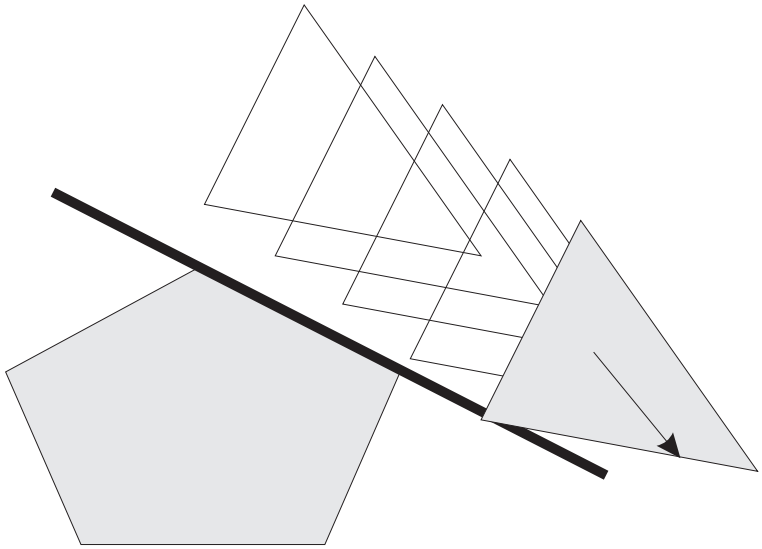


Figura 3.8: Un plano que deja de ser separador.

caras/aristas/vértices coincidentes con dicho plano. El Algoritmo 3.3 muestra un esbozo del proceso completo. Se hace uso de una función *AgruparCaras* que devuelve todas las caras/aristas/vértices que se encuentran en el plano separador (Algoritmo 3.4). Cabe destacar que todas las comprobaciones de coincidencia/intersección se deben realizar bajo una tolerancia TOL y que, por tanto, en lugar de comprobar que un vértice se encuentra dentro de una determinada cara debemos comprobar que se encuentra *cerca* de ésta. Esto conlleva el cálculo de distancia de un punto a un plano (para colisiones vertice/cara) y la distancia más cercana entre dos rectas (para colisiones arista/arista). Por motivos de simplicidad este tipo de detalles se omiten en el Algoritmo 3.3.

El coste de este algoritmo varía según sea o no necesaria la búsqueda de un plano separador. En el caso de no ser necesaria y no haber ningún contacto, sólo se habría comprobado si el plano actual es separador, lo cual conlleva un coste del $\mathcal{O}(V_A + V_B)$, donde V_A y V_B son el número de vértices de cada objeto.

Existen otros métodos de detección de colisiones, como los algoritmos de *Lin-Canny* [Lin93] y *GJK* [GJK88], que calcula la distancia mínima entre dos poliedros convexos que no interpenetran. El algoritmo *V-Clip* [Mir97] está basado en el primero, superando algunas de sus limitaciones. Para poliedros no convexos podemos citar [GBF03].

si $(planoSeparador = \texttt{NULL}) \vee_c \neg(EsSeparador(planoSeparador))$ **entonces**
    $planoSeparador := BuscarPlanoSeparador(A, B)$
fsi

si $planoSeparador = \texttt{NULL}$ **entonces**
    devolver INTERPENETRACIÓN
sino
    $listaContactos := []$
    $< caras_A, aristas_A, vertices_A >:= AgruparCaras(A, planoSeparador)$
    $< caras_B, aristas_B, vertices_B >:= AgruparCaras(B, planoSeparador)$

    para cada cara c_A de $caras_A$ **hacer**
        para cada vertice v_B de $vertices_B$ **hacer**
            si v_B está en c_A **hacer**
                Añadir contacto entre v_B y c_A a $listaContactos$
            fsi
        fpara
    fpara

    para cada cara c_B de $caras_B$ **hacer**
        para cada vertice v_A de $vertices_A$ **hacer**
            si v_A está en c_B **hacer**
                Añadir contacto entre v_A y c_B a $listaContactos$
            fsi
        fpara
    fpara

    para cada arista e_A de $aristas_A$ **hacer**
        para cada arista e_B de $aristas_B$ **hacer**
            si e_A interseca con e_B **hacer**
                Añadir contacto entre e_A y e_B a $listaContactos$
            fsi
        fpara
    fpara
    devolver $listaContactos$
fsi

Algoritmo 3.3: Colisiones entre mallas

---

proc *AgruparCaras*(*Objeto*, *Plano*)
    caras := {}
    aristas := {}
    vertices := {}
    para cada cara *c* de *Objeto* **hacer**
        n := número de vértices que tiene *c* en *Plano*
        casos
            $n = 0$ → **nada**
            $n = 1$ → Añadir el vértice a *vertices*
            $n = 2$ → Añadir los dos vértices a *vertices*
                Añadir la arista que los une a *aristas*
            $n = 3$ → Añadir los tres vértices a *vertices*
                Añadir las tres aristas a *aristas*
                Añadir la cara a *caras*
        fcasos
    fpara

    devolver $< caras, aristas, vertices >$
fproc

Algoritmo 3.4: Detección de caras/aristas/vértices en el plano separador

3.8. Detalles de implementación

Toda la familia de algoritmos explicados en este capítulo ha sido organizada haciendo uso del patrón *Estrategia* [GHJV95]. Cada algoritmo queda encapsulado en una clase descendiente de la clase abstracta **rbsAlgoritmoColision**. Esto permite que el usuario de la API pueda, por un lado, ampliar el conjunto de algoritmos de colisión entre geometrías no contempladas en la jerarquía de la Figura 1.7 y, por otro lado, disponer de varios algoritmos intercambiables para un mismo tipo de geometría.

La clase **rbsAlgoritmoColision** contiene dos métodos abstractos, que cada algoritmo completo debe definir:

```
bool CalcularColision(std::vector<rbsContacto> &contactos)
void GetContactosCercanos(std::vector<rbsContacto> &contactos)
```

El último de ellos se utiliza en casos muy específicos, en los que el método de bisección no es capaz de determinar el tiempo exacto de colisión. Esto se explicará con más detalle en la Sección 5.2.1.

El método `CalcularColision` es el encargado de calcular los contactos que se producen entre dos instancias de la clase **rbsObjeto**, que son atributos de la clase **rbsAlgoritmoColision**. Este método debe comportarse de la siguiente forma:

- Si los objetos están **interpenetrando**, debe devolver `true`.
- Si los objetos están en **contacto**, debe devolver `false`, almacenando en el vector `contactos` pasado como referencia las distintas instancias de **rbsContacto**.
- En otro caso, debe devolver `false`, dejando el vector `contactos` vacío.

Una vez que hayamos creado una instancia del algoritmo correspondiente, debemos registrarla en uno de los objetos cuya colisión se va a detectar. Para ello utilizamos el método de **rbsObjeto**:

```
void InsertarDetectorColisiones(rbsObjeto* objeto,
                                rbsAlgoritmoColision* detector)
```

Por ejemplo, la sentencia:

```
A->InsertarDetectorColisiones(B, alg);
```

indica a la API que a partir de ese momento utilice el algoritmo `alg` para detectar colisiones entre los objetos `A` y `B`. Es equivalente a utilizar:

```
B->InsertarDetectorColisiones(A, alg);
```

Capítulo 4

Respuesta a las colisiones. Impulso.

4.1. Introducción

Mediante las rutinas de detección vistas anteriormente obtenemos todos los contactos que se producen desde el tiempo actual hasta que haya transcurrido un paso de simulación. Una vez que se han detectado todos los puntos de contacto entre dos objetos de la escena, es necesario tratarlos y aplicar una respuesta adecuada. Esta respuesta debe aplicarse en un único punto de contacto para cada objeto. Por tanto, necesitaremos una forma de decidir cuál es ese punto a partir de todos los contactos. Se comentará en la Sección 4.3.

4.2. Definición y cálculo del impulso

Recordemos que un contacto tiene como elementos principales una referencia a cada uno de los dos cuerpos que colisionan (A y B), el punto de contacto p y la normal de contacto $\mathbf{n}$ (que debe salir de B y apuntar a A).

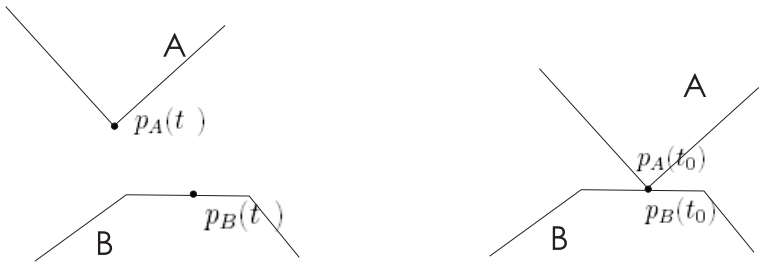


Figura 4.1: Puntos p_A y p_B de contacto.

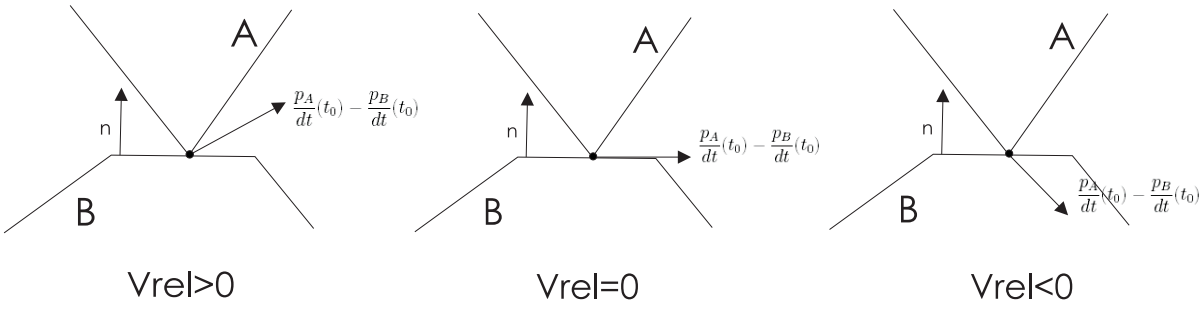


Figura 4.2: Velocidad relativa entre dos objetos.

Cada contacto se producirá en un determinado instante de tiempo t_0. En ese instante un punto del objeto A ($p_A(t_0)$) y otro del objeto B ($p_B(t_0)$) serán iguales a p (Figura 4.1). La velocidad del punto en cada uno de estos objetos en el tiempo t_0 se calcula como se vió en la sección 1.2. Expresamos dicha velocidad como la derivada de la posición. Por tanto:

$$\frac{dp}{dt} = \mathbf{v}(t) + \omega(t) \times (p(t) - \mathbf{x}(t)) \tag{4.1}$$

Donde: p es el punto del que queremos conocer la velocidad, $\mathbf{v}$ es la velocidad lineal del objeto, ω es su velocidad angular y $\mathbf{x}$ es su posición.

Aplicando la Ecuación (4.1) a los puntos $p_A(t_0)$ y $p_B(t_0)$, tenemos:

$$\frac{dp_A}{dt}(t_0) = \mathbf{v}_A(t_0) + \omega_A(t_0) \times (p_A(t_0) - \mathbf{x}_A(t_0)) \tag{4.2}$$

$$\frac{dp_B}{dt}(t_0) = \mathbf{v}_B(t_0) + \omega_B(t_0) \times (p_B(t_0) - \mathbf{x}_B(t_0)) \tag{4.3}$$

Nos interesa conocer el valor de la velocidad relativa entre los puntos $p_A(t_0)$ y $p_B(t_0)$. Ésta se define como:

$$\mathbf{v}_{rel} = \mathbf{n}(t_0) \left(\frac{dp_A}{dt}(t_0) - \frac{dp_B}{dt}(t_0) \right) \tag{4.4}$$

donde $\mathbf{n}(t_0)$ es la normal de contacto $\mathbf{n}$.

El resultado de v_{rel} es un escalar. Según su valor se pueden dar tres casos (Figura 4.2):

- $v_{rel} > 0$. En este caso los cuerpos se están alejando y por tanto, no habrá que aplicar ningun tipo de impulso.
- $v_{rel} = 0$. En este caso los cuerpos están en resting contact. Veremos más adelante (Capítulo 6) cómo se debe actuar.

- $v_{rel} < 0$. Este es el caso que nos interesa. Los cuerpos colisionan (se están acercando) y por tanto será necesario modificar sus velocidades para que no penetren.

En este último caso la velocidad de los cuerpos debe ser modificada instantáneamente; esto no lo podemos conseguir aplicando fuerzas. Para producir este cambio instantáneo de velocidad aplicaremos un impulso J. El impulso se puede considerar como una gran fuerza que actúa durante un corto período de tiempo:

$$\mathbf{J} = \mathbf{F}\Delta t$$

El cambio de velocidad que produce $\mathbf{J}$ puede considerarse como el cambio de velocidad que produciría $\mathbf{F}$ si la aplicásemos durante un intervalo de tiempo Δt.

La velocidad tiene una componente lineal y una angular. Vamos a ver el cambio que se produce en cada componente. El cambio en la velocidad lineal que produce $\mathbf{J}$ si se aplica a un cuerpo de masa M es:

$$\Delta\mathbf{v} = \frac{\mathbf{J}}{M} \tag{4.5}$$

Esto equivale a que el cambio del momento lineal es $\Delta\mathbf{P} = \mathbf{J}$. Al ser aplicado en un punto p, al igual que las fuerzas, produce un torque. Su valor es:

$$\Gamma_{impulso} = (p - \mathbf{x}(t)) \times \mathbf{J}$$

$\Gamma_{impulso}$ produce un cambio en el momento angular: $\Delta\mathbf{L} = \Gamma_{impulso}$. Este cambio afecta a la velocidad angular de la siguiente manera:

$$\Delta\omega = I^{-1}(t_0)\Gamma_{impulso} \tag{4.6}$$

Suponemos que el impulso es aplicado en t_0.

Ya sabemos cómo afecta un impulso cuando los cuerpos colisionan, pero todavía no sabemos cómo calcularlo. El impulso, en el caso de que no consideremos fricción, tiene la dirección de la normal de contacto $\mathbf{n}(t_0)$. Por tanto:

$$\mathbf{J} = j\mathbf{n}(t_0)$$

donde j es la magnitud del impulso. Consideraremos que el impulso actúa de forma positiva sobre el cuerpo A $(+j\mathbf{n}(t_0))$, mientras que en B actuará negativamente $(-j\mathbf{n}(t_0))$. Ver Figura 4.3.

Para las siguientes operaciones utilizaremos la siguiente notación: $\left(\frac{dp_A}{dt}(t_0)\right)^-$ es la velocidad del punto de contacto de A antes de aplicar el impulso, $\left(\frac{dp_A}{dt}(t_0)\right)^+$ es la velocidad

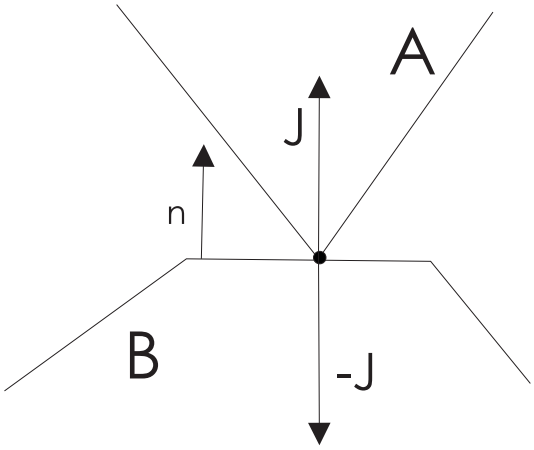


Figura 4.3: Dirección del impulso aplicado a dos cuerpos que colisionan.

después del aplicarlo. Del mismo modo definimos $\left(\frac{dp_B}{dt}(t_0)\right)^-$ y $\left(\frac{dp_B}{dt}(t_0)\right)^+$. Con esta notación expresamos la velocidad relativa antes y después del impulso.

Antes del impulso:

$$v_{rel}^- = \mathbf{n}(t_0)\left(\left(\frac{dp_A}{dt}(t_0)\right)^- - \left(\frac{dp_B}{dt}(t_0)\right)^-\right) \tag{4.7}$$

Después del impulso:

$$v_{rel}^+ = \mathbf{n}(t_0)\left(\left(\frac{dp_A}{dt}(t_0)\right)^+ - \left(\frac{dp_B}{dt}(t_0)\right)^+\right) \tag{4.8}$$

Según leyes empíricas de colisión sin fricción se tiene que:

$$v_{rel}^+ = -\epsilon v_{rel}^- \tag{4.9}$$

ϵ se conoce como **coeficiente de restitución** y su valor debe ser $0 \leq \epsilon \leq 1$. En caso de que $\epsilon = 1$ se tiene que el la energía cinética tras la colisión se conserva. Si $\epsilon = 0$ la velocidad de los cuerpos tras la colisión es nula.

Para determinar el impulso que debemos aplicar tendremos que calcular el valor de j. En el resto del capítulo nos dedicaremos a ello. La idea consiste en expresar v_{rel}^+ en función de v_{rel}^- y j, para poder despejarlo de la ecuación (4.9) con facilidad. Para empezar calculamos la velocidad de $p_A(t_0)$ después de la colisión en función de su velocidad antes de la colisión; idem para $p_B(t_0)$. De esta forma podremos sustituir dichos valores en la ecuación (4.7).

$$\left(\frac{dp_A}{dt}(t_0)\right)^+ = \mathbf{v}_A^+ + \omega_A^+ \times \mathbf{r}_A \tag{4.10}$$

donde

$$\mathbf{r}_A = p - \mathbf{x}_A(t_0)$$

$\mathbf{v}_A^+$ y ω_A^+ son las velocidades tras aplicar el impulso. Por tanto, por (4.5) y (4.6):

$$\mathbf{v}_A^+ = \mathbf{v}_A^- + \Delta\mathbf{v} = \mathbf{v}_A^- + \frac{j\mathbf{n}(t_0)}{M_A} \tag{4.11}$$

$$\omega_A^+ = \omega_A^- + \Delta\omega = \omega_A^- + I_A^{-1}(t_0)\Gamma_{impulso} = \omega_A^- + I_A^{-1}(t_0)(\mathbf{r}_A \times j\mathbf{n}(t_0)) \tag{4.12}$$

Sustituyendo en la ecuación (4.10) tenemos que:

$$\begin{aligned}\left(\frac{dp_A}{dt}(t_0)\right)^+ &= \left(\mathbf{v}_A^- + \frac{j\mathbf{n}(t_0)}{M_A}\right) + \left(\omega_A^- + I_A^{-1}(t_0)(\mathbf{r}_A \times j\mathbf{n}(t_0))\right) \times \mathbf{r}_A \\ &= \mathbf{v}_A^- + \omega_A^- \times \mathbf{r}_A + \left(\frac{j\mathbf{n}(t_0)}{M_A}\right) + \left(I_A^{-1}(t_0)(\mathbf{r}_A \times j\mathbf{n}(t_0))\right) \times \mathbf{r}_A \\ &= \left(\frac{dp_A}{dt}(t_0)\right)^- + j\left(\frac{\mathbf{n}(t_0)}{M_A} + I_A^{-1}(t_0)(\mathbf{r}_A \times \mathbf{n}(t_0)) \times \mathbf{r}_A\right)\end{aligned}$$

Realizamos los mismos cálculos para la velocidad del punto de B, pero teniendo en cuenta que el impulso tiene signo contrario ($-j\mathbf{n}(t_0)$):

$$\left(\frac{dp_B}{dt}(t_0)\right)^+ = \left(\frac{dp_B}{dt}(t_0)\right)^- - j\left(\frac{\mathbf{n}(t_0)}{M_B} + I_B^{-1}(t_0)(\mathbf{r}_B \times \mathbf{n}(t_0)) \times \mathbf{r}_B\right) \tag{4.13}$$

Usando estas dos últimas ecuaciones calculamos su resta, para sustituirla en (4.7).

$$\left(\frac{dp_A}{dt}(t_0)\right)^+ - \left(\frac{dp_B}{dt}(t_0)\right)^+ = \left(\left(\frac{dp_A}{dt}(t_0)\right)^- - \left(\frac{dp_B}{dt}(t_0)\right)^-\right) +$$

$$j\left(\frac{\mathbf{n}(t_0)}{M_A} + \frac{\mathbf{n}(t_0)}{M_B} + \left(I_A^{-1}(t_0)(\mathbf{r}_A \times \mathbf{n}(t_0))\right) \times \mathbf{r}_A + \left(I_B^{-1}(t_0)(\mathbf{r}_B \times \mathbf{n}(t_0))\right) \times \mathbf{r}_B\right)$$

Calculamos v_{ref}^+. Debido a que el vector $\mathbf{n}(t_0)$ tiene módulo 1, entonces $\mathbf{n}(t_0) \cdot \mathbf{n}(t_0) = 1$.

$$\begin{aligned}v_{rel}^+ &= \mathbf{n}(t_0)\left(\left(\frac{dp_A}{dt}(t_0)\right)^+ - \left(\frac{dp_B}{dt}(t_0)\right)^+\right) \\ &= \mathbf{n}(t_0)\left(\left(\left(\frac{dp_A}{dt}(t_0)\right)^- - \left(\frac{dp_B}{dt}(t_0)\right)^-\right)\right. \\ &+ \left. j\left(\frac{\mathbf{n}(t_0)}{M_A} + \frac{\mathbf{n}(t_0)}{M_B} + \left(I_A^{-1}(t_0)(\mathbf{r}_A \times \mathbf{n}(t_0))\right) \times \mathbf{r}_A + \left(I_B^{-1}(t_0)(\mathbf{r}_B \times \mathbf{n}(t_0))\right) \times \mathbf{r}_B\right)\right) \\ &= v_{rel}^- + j\left(\frac{1}{M_A} + \frac{1}{M_B} + \left(I_A^{-1}(t_0)(\mathbf{r}_A \times \mathbf{n}(t_0))\right) \times \mathbf{r}_A + \left(I_B^{-1}(t_0)(\mathbf{r}_B \times \mathbf{n}(t_0))\right) \times \mathbf{r}_B\right)\end{aligned}$$

Hemos conseguido expresar v_{rel}^+ en función de j y v_{rel}^-. Sustituiremos v_{rel}^+ en (4.9):

$$v_{rel}^- + j\left(\frac{1}{M_A} + \frac{1}{M_B} + \left(I_A^{-1}(t_0)(\mathbf{r}_A \times \mathbf{n}(t_0))\right) \times \mathbf{r}_A + \left(I_B^{-1}(t_0)(\mathbf{r}_B \times \mathbf{n}(t_0))\right) \times \mathbf{r}_B\right) = -\epsilon v_{rel}^- \tag{4.14}$$

Por último, podemos despejar j:

$$j = \frac{-(1+\epsilon)v_{rel}^-}{\left(\frac{1}{M_A} + \frac{1}{M_B} + \left(I_A^{-1}(t_0)(\mathbf{r}_A \times \mathbf{n}(t_0))\right) \times \mathbf{r}_A + \left(I_B^{-1}(t_0)(\mathbf{r}_B \times \mathbf{n}(t_0))\right) \times \mathbf{r}_B\right)} \tag{4.15}$$

4.3. Detalles de implementación

Para el cálculo y aplicación del Impulso existe en **rbsEscena** un método que se encarga de ello:

```
void rbsEscena::Impulso (rbsColision* colision)
```

La clase **rbsColision** guarda la siguiente información:

- Las referencias a los dos **objetos** que están en colisión. Estos objetos serán referidos como A y B.
- El **punto de colisión**. Se especifica en coordenadas globales de la escena. Si sólo hay un contacto, es precisamente igual al punto de contacto. Si hay más de uno, se deberá tener una estrategia que reúna todos los puntos de contacto en uno, ya que el impulso sólo se debe aplicar a cada objeto en un punto. La estrategia utilizada consiste en calcular el punto intermedio de la caja mínima alineada con los ejes que agrupa a todos los puntos de contacto.
- **Vector normal de colisión**. Por convenio, este vector debe apuntar hacia el objeto A. Debe ser igual que la normal de todos los contactos.
- **Lista de contactos**. Contactos correspondientes a esta colisión.
- **Tiempo**. Tiempo estimado en el que se producirá la colisión

Este método será llamado en el bucle de simulación, que se detallará en el Capítulo 5.

Con respecto al coeficiente de restitución, cada objeto lleva una tabla que almacena todos los ϵ que se utilizarán en el cálculo del impulso en colisiones con los demás objetos. Para asociar un par de objetos con un determinado coeficiente, podemos utilizar el siguiente método de **rbsObjeto**:

```
void rbsObjeto::InsertarCoeficienteRestitucion(rbsObjeto* objeto,
                                               TIPOREAL coeficiente)
```

La siguiente sentencia:

```
A->InsertarCoeficienteRestitucion(B, 0.4);
```

que es equivalente a:

```
B->InsertarCoeficienteRestitucion(A, 0.4);
```

Provoca que a partir del momento el cálculo del impulso en las colisiones entre A y B se realice con un $\epsilon = 0{,}4$.

Capítulo 5

Integración de colisiones en la simulación

5.1. Bounding Boxes

5.1.1. Introducción

La técnica de las *bounding boxes* (cajas acotantes) consiste en delimitar los objetos mediante paralelepípedos alineados con los ejes de la escena para **descartar** con facilidad ciertas parejas de objetos que no podrán colisionar. La forma de usarla es sencilla: Antes de aplicar los algoritmos de colisión descritos en el Capítulo 3 a un par de objetos, comprobamos si se produce colisión entre sus respectivas *bounding boxes*. Si no se produce dicha colisión, entonces ese par de objetos **queda descartado** como candidato a colisionar. De esta forma evitaremos recurrir a estos algoritmos sobre objetos que sabemos que no van a colisionar, y por lo tanto, ganaremos eficiencia. Por el contrario, si hay colisión entre las bounding boxes no sabemos si los cuerpos colisionan o no, por lo que debemos continuar el análisis con los algoritmos del Capítulo 3.

5.1.2. Algoritmo de Barrido

Para abarcar el problema de detección (ver [Bar97c]) consideraremos en principio *bounding boxes* de una dimensión, es decir, intervalos de la forma $[i, f] \subset \mathbb{R}$. Supongamos que tenemos n intervalos, siendo el $j-$ésimo intervalo $[i_j, f_j]$. Nuestro objetivo es detectar los solapamientos entre estos intervalos.

En primer lugar ordenaremos la lista de todos los extremos de los intervalos (los i_j y f_j) y creamos una lista vacía de *intervalos activos*. A continuación se realiza un *barrido* desde

el intervalo de menor coordenada al de mayor. Se tratará cada extremo de la siguiente forma:

- Si es un extremo de **inicio de intervalo**, habrá solapamiento entre el intervalo al que pertenece este extremo y todos los intervalos que se encuentren en la lista de intervalos activos. Además añadimos el intervalo actual a la lista de intervalos activos.
- Si es un extremo de **fin de intervalo**, eliminaremos de la lista de intervalos activos el intervalo al que pertenece este extremo.

Si tenemos n intervalos y se detectan k solapamientos, el coste del algoritmo es del $\mathcal{O}(n\ log(n))$ para la ordenación de los extremos, $\mathcal{O}(n)$ para el barrido y $\mathcal{O}(k)$ para almacenar los pares solapantes. Esto da un coste total de $\mathcal{O}(n\ log(n) + k)$.

No obstante, el coste puede mejorarse teniendo en cuenta que la posición relativa de los intervalos varía con poca frecuencia. En lugar de realizar la ordenación de los extremos en cada barrido utilizando $MergeSort$, podemos utilizar otras alternativas como $BubbleSort$ que resultan más eficientes cuando estamos tratando vectores *casi* ordenados. Con esta mejora, el coste llega a $\mathcal{O}(n + k)$, que degenera en $\mathcal{O}(n^2)$ cuando todos los intervalos se solapan entre ellos.

5.1.3. Detalles de implementación

Cada tipo de geometría de la que disponemos (ver Figura 1.7) tiene asociada su *bounding box*. Ésta es creada en los constructores de cada geometría, ya que depende de la forma del objeto. Cada *bounding box* es representada mediante un array de 6 números reales, que representan sus límites (límite inferior sobre el eje x, límite superior sobre el eje x y del mismo modo para los límites sobre los ejes y,z).

Para su gestión e integración en la escena, disponemos de la clase **rbsGestorBB**. La escena contendrá tres **rbsGestorBB**, uno para cada eje coordenado (x,y,z). Cada gestor mantiene una lista de *bordes bounding box*, que es doblemente enlazada y ordenada de forma ascendente por el valor de las coordenadas de los puntos límite de los *bordes bounding box* en el eje que les corresponde. Además, cada *borde bounding box* contendrá una referencia al objeto al cual pertenece. Cada objeto que hay en la escena tendrá dos entradas en la lista ordenada, una para el valor mínimo (límite inferior) de su *bounding box* y otra para el valor máximo (límite superior), en ese eje. De esta forma, la referencia al objeto funciona como identificador único de intervalos (identifica su intervalo, que se extiende desde la coordenada del borde de inicio hasta la coordenada del borde de fin).

Así pues, cada objeto tiene asociados 6 *bordes bounding box*, dos en cada **rbsGestorBB**. (Figura 5.1)

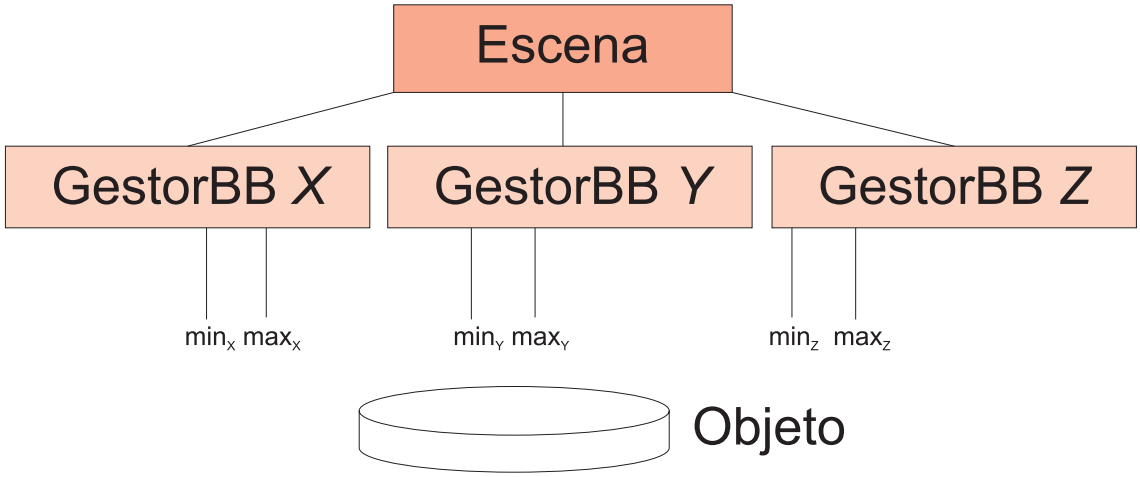


Figura 5.1: Gestores de bounding boxes asociados la escena y bounding boxes asociadas a cada objeto.

En la escena, a la hora de calcular el siguiente estado en la simulación, se actualizará la posición de cada objeto, y en consecuencia también se deben actualizar cada uno de los bordes (límites) de la *bounding box* del objeto, manteniendo las listas de sus gestores ordenadas crecientemente por sus coordenadas.

Por otra parte, dos *bounding boxes* se solaparán si y sólo si sus coordenadas se solapan **para cada uno de los tres ejes**. Podemos encontrar en ejemplo en dos dimensiones en la Figura 5.2: los objetos 3 y 1 se solapan ya que tanto los intervalos $[x1, x1']$ y $[x3, x3']$ como los intervalos $[y1, y1']$ y $[y3, y3']$ solapan. Sin embargo, los objetos 3 y 2 no se solapan ya que los intervalos $[y3, y3']$ y $[y2, y2']$ se solapan, pero no los intervalos $[x3, x3']$ y $[x2, x2']$.

En la clase **rbsGestorBB** tenemos un método llamado `ObtenerSolapamientos()`, que devuelve el conjunto de pares de objetos cuyas *bounding boxes* se solapan. Como se ha podido ver, en el proceso **todas las colisiones entre *bounding boxes* han sido detectadas simultáneamente**.

5.2. Método de bisección

Una vez descartados los objetos que no intersecarán en el próximo paso de simulación, podemos examinar cada par de objetos devuelto por la comprobación de *bounding boxes* y utilizar uno de los algoritmos de colisión comentados en el Capítulo 3 para ver si los objetos colisionan realmente.

Sin embargo, para este proceso no sólo resulta relevante la presencia o ausencia de colisiones; también es necesario determinar el **tiempo de colisión** entre cada par de objetos.

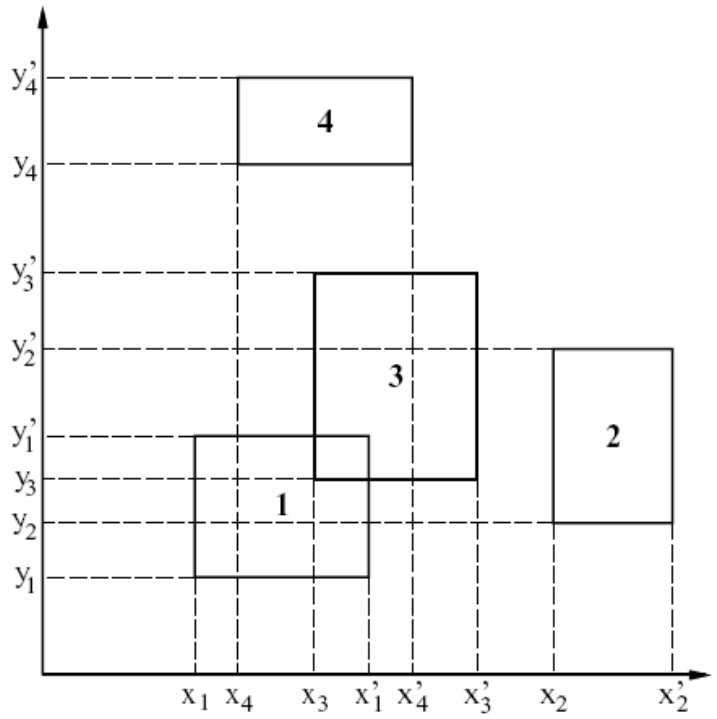


Figura 5.2: Intersecciones de *bounding box* en dos dimensiones. Dos *bounding boxes* se solaparán si y sólo si se solapan los intervalos dados por sus límites sobre el eje x y el eje y.

El método que proponemos en esta sección para este fin recibe el nombre de **bisección temporal**.

Sea t_0 el tiempo de escena actual y Δt el tiempo de paso de simulación que se quiere avanzar. Sean A y B dos objetos. Si avanzamos el estado de A y B a un tiempo $t_0 + \Delta t$ y comprobamos mediante el detector de colisiones correspondiente que *no* interpenetran, podemos ignorarlos y repetir el método para el siguiente par de objetos. En caso contrario, tendremos que **restaurar** el estado de los objetos a t_0 y volver a avanzarlo, pero esta vez sólo la mitad del paso (esto es, hasta $t_0 + \Delta t/2$). En este punto:

- Si A y B están en contacto, pero sin llegar a interpenetrar, ya conocemos el tiempo de colisión.
- Si A y B están interpenetrando, tendremos que retroceder otra vez al tiempo t_0 y volver a avanzar el estado hasta $t_0 + \Delta t/4$ y repetir el proceso. El tiempo de colisión queda acotado en el intervalo $(t_0, t_0 + \Delta t/2)$
- Si A y B se encuentran separados, tendremos que avanzar hasta $t_0 + 3\Delta t/4$ y repetir el proceso. El tiempo de colisión queda acotado en el intervalo $(t_0 + \Delta t/2, t_0 + \Delta t)$.

De este modo podemos acotar sucesivamente el tiempo de colisión hasta que el algoritmo de colisión utilizado para A y B devuelva uno o más contactos. En el Algoritmo 5.1 se describe el proceso completo. Aunque no se detalla en este pseudocódigo, el algoritmo realmente implementado devuelve, además del tiempo de colisión, los contactos. En la Figura 5.3 se muestra un ejemplo de cálculo de tiempo de colisión para una esfera y una pared fija.

5.2.1. Contactos de emergencia

El bucle del Algoritmo 5.1 finaliza cuando se ha detectado un contacto entre los objetos, o bien cuando t_{inicio} y t_{fin} llegan a igualarse (bajo un umbral de tolerancia TOL). En este último caso hemos llegado a un intervalo $[t_{inicio}, t_{fin}]$ muy reducido y el algoritmo de colisión correspondiente no parece detectar ningún contacto.

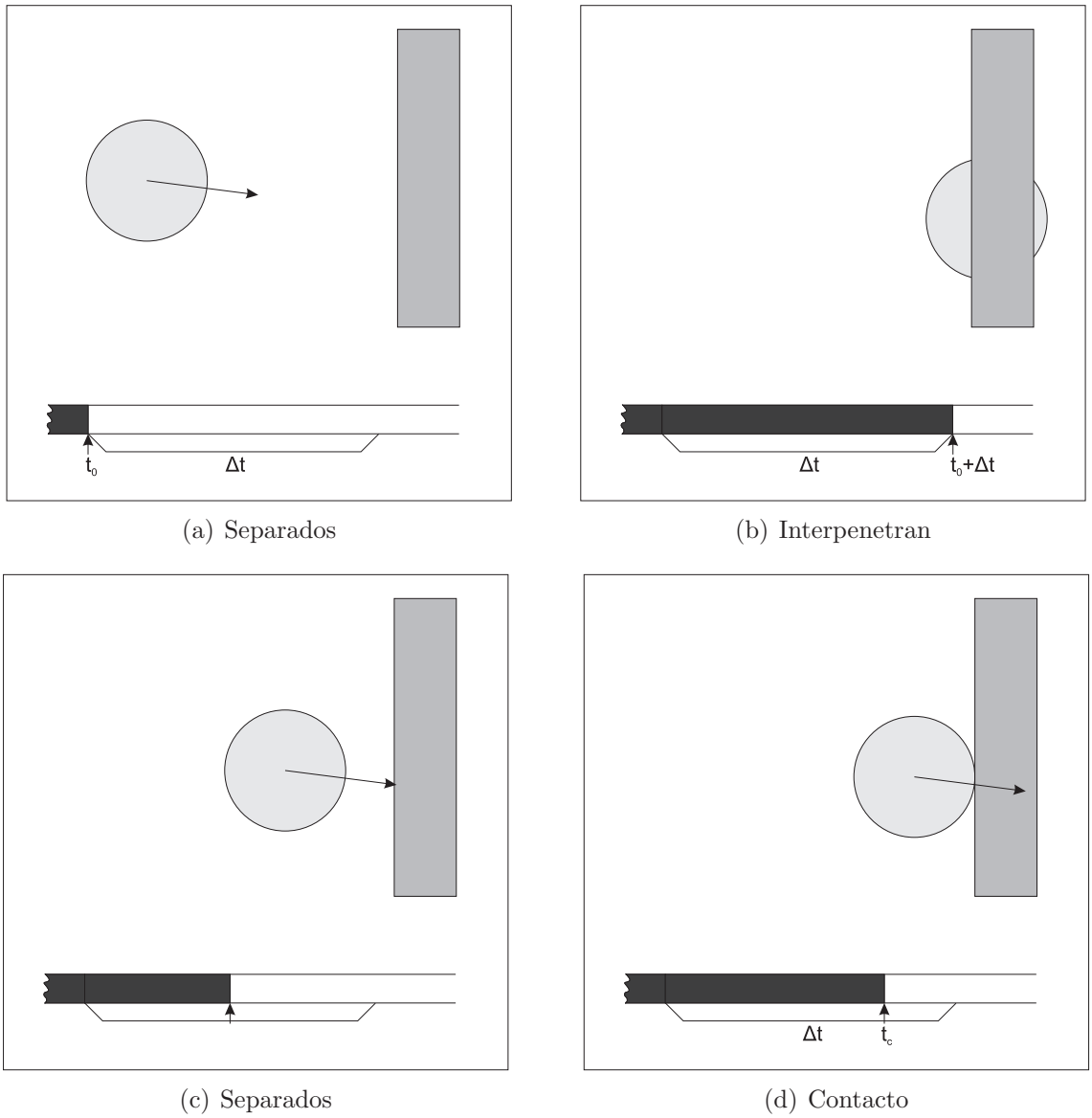
Esto puede deberse, por una parte, a una mala implementación del algoritmo de colisión utilizado. Un usuario de la API que extienda la clase **rbsAlgoritmoColision** codificando un algoritmo que no devuelva nunca ningún contacto, y utilizándolo luego para los objetos A y B hará que el método de bisección llegue a esta situación. Por otro lado, y aún estando bien implementado el algoritmo, los errores de redondeo a lo largo de los cálculos realizados puede provocar que el algoritmo no consiga nunca detectar un contacto bajo un determinado umbral de tolerancia.

---

proc $CalcularTiempoColision(A, B, t_0, \Delta t)$
$A.ApilarEstado();\quad B.ApilarEstado();$
$A.SiguienteEstado(\Delta t);\quad B.SiguienteEstado(\Delta t);$
casos
A y B interpenetran $\rightarrow$ **nada**
A y B contactan $\rightarrow$ $A.DesapilarEstado();\quad B.DesapilarEstado();$
devolver $t_0 + \Delta t$
A y B están separados $\rightarrow$ **devolver** `NULL`
fcasos
$t_{inicio} = t_0$
$t_{fin} = t_0 + \Delta t$
mientras $t_{fin} - t_{inicio} > TOL$ **hacer**
$t_{mitad} = (t_{fin} - t_{inicio})/2$
$A.ApilarEstado();\quad B.ApilarEstado();$
$A.SiguienteEstado(t_{mitad});$
$B.SiguienteEstado(t_{mitad});$
casos
A y B interpenetran $\rightarrow$ $t_{fin} = t_{mitad}$
A y B contactan $\rightarrow$ $A.DesapilarEstado();\quad B.DesapilarEstado();$
devolver t_{mitad}
A y B están separados $\rightarrow$ $t_{inicio} = t_{mitad}$
fcasos
$A.DesapilarEstado();\quad B.DesapilarEstado();$
fmientras

Tratar contactos de emergencia y devolverlos, con tiempo de colisión t_{inicio}
fproc

Algoritmo 5.1: Método de bisección



(a) Separados

(b) Interpenetran

(c) Separados

(d) Contacto

Figura 5.3: Cálculo del tiempo de colisión para una esfera y una pared fija.

$rbsEscena :: SiguienteEstado(\Delta t)$
  $t_{restante} := \Delta t$
  mientras $t_{restante} \neq 0$ **hacer**
    $t_{avanzado} := AvanzarHastaColision(t_{restante})$
    $escena \rightarrow tiempo := escena \rightarrow tiempo + t_{avanzado}$
    $t_{restante} := t_{restante} - t_{avanzado}$
  fmientras

Algoritmo 5.2: Bucle principal de la escena con colisiones.

En estos casos se llamará al método `GetContactosCercanos()` del algoritmo de colisión correspondiente. Con esta llamada estamos indicando al algoritmo de colisión que los objetos **deberían** estar en contacto y que se espera que nos proporcione como mínimo un punto de contacto y una normal.

5.3. Bucle de simulación

Una vez descritas las técnicas y algoritmos relevantes, podemos realizar una descripción del bucle de simulación de la escena, que integra:

- **Estado** de los objetos y fuerzas aplicadas (Capítulo 1). Comprende el bucle de simulación inicial descrito en la Sección 1.7.7.
- **Algoritmos** de detección de colisiones (Capítulo 3).
- Cálculo del **Impulso** (Capítulo 4).
- Detección rápida de colisiones mediante ***bounding boxes*** (Sección 5.1).
- Método de **bisección temporal** (Sección 5.2).

Sea t_0 el tiempo de escena actual y Δt el tamaño de paso. Definiremos un método auxiliar $AvanzarHastaColision(\Delta t)$ que nos permita avanzar la simulación hasta que se produzca la próxima colisión entre dos objetos cualesquiera de la escena, o en el caso de no producirse ninguna colisión, hasta que se haya avanzado un tiempo Δt. El bucle principal de la escena consistirá en llamar repetidamente a este método hasta que se haya simulado todo el intervalo Δt (Algoritmo 5.2).

Hemos trasladado el problema al método *AvanzarHastaColision* (Algoritmo 5.3). Si Δt es el intervalo pasado como parámetro, avanzamos todos los objetos un tiempo Δt para realizar la comprobación mediante *bounding boxes*. Tras volver a restaurar el estado de los objetos, nuestro objetivo es recorrer todos los pares de objetos no descartados en busca de la colisión más cercana, que se producirá en un tiempo t_{minimo}. Si varias colisiones ocurren en el tiempo t_{minimo}, todas ellas serán tratadas simultáneamente.

El método *BuscarColisionesMasCercanas* (Algoritmo 5.4) es el encargado de aplicar el método de bisección a todos los pares de objetos no descartados. En este punto cabe destacar una serie de consideraciones:

- Una vez encontrada una colisión menor que el t_{minimo} actual, **las siguientes llamadas al método de bisección sólo se realizan hasta dicho** t_{minimo}, pues a partir de ese momento nos interesarán sólo las colisiones cuyo $t_{colision}$ sea menor o igual (las que tengan el $t_{colision}$ mayor no se añadirían a *listaColisiones*). Puede entenderse esto como una especie de "poda"

- Aunque no se indicó explicitamente en el Algoritmo 5.1, recordemos que el método *CalcularTiempoColision* devolvía, además del tiempo de colisión, los contactos producidos.

- Cuando encontramos una colisión que se produce en un tiempo menor que el t_{minimo} encontrado hasta ahora, se vacía la lista de colisiones encontradas hasta el momento. Estas colisiones que son borradas podrían almacenarse en una estructura auxiliar para que en una próxima llamada a *CalcularTiempoColision* dentro del mismo paso de simulación no tengan que volver a calcularse. Para no complicar el pseudocódigo en exceso no hemos incluido esta optimización, pero si se aplica, se debe tener en cuenta que cuando **se aplica un impulso a dos objetos, se deben borrar todas las colisiones que involucren a uno de estos dos objetos de la estructura auxiliar**, ya que las colisiones que se encuentran allí dejan de ser válidas, al haber cambiado los objetos de dirección.

- El tratamiento de las colisiones que se producirán en un tiempo más cercano (método *AvanzarHastaColision*) se debe repetir hasta que todos los pares de objetos de cada colisión se estén alejando o estén en *resting contact*. De ello se encarga el bucle que acaba cuando *hayColisiones* = `FALSE`. El motivo de esto es que una vez que se aplica un impulso tenemos que reexaminar la lista de contactos, ya que algunos cuerpos que se estaban alejando podrían estar ahora en colisión y viceversa.

$rbsEscena :: AvanzarHastaColision(\Delta t)$
apilar el estado de todos los objetos de la escena
avanzar el estado de todos los objetos de la escena en un tiempo Δt
$paresBB := SolapamientosPorBoundingBox()$
desapilar el estado de todos los objetos de la escena

$\langle colisiones, t_{minimo} \rangle := BuscarColisionesMasCercanas(paresBB, \Delta t)$

avanzar todos los objetos de la escena un tiempo t_{minimo}

repetir
  $hayColisiones =$ `FALSE`
  para cada colision c de $colisiones$ **hacer**
    si los objetos involucrados en c se están acercando **entonces**
      calcular y aplicar impulso a los objetos participantes en c
      $hayColisiones =$ `TRUE`
    fsi
  fpara
hasta que $hayColisiones =$ `FALSE`

devolver t_{minimo}

Algoritmo 5.3: Método $AvanzarHastaColision$

$rbsEscena :: BuscarColisionesMasCercanas(paresBB, \Delta t)$
    $t_{minimo} := \Delta t$
    $listaColisiones := [\,]$
    para cada par $\langle A, B\rangle$ de $paresBB$ **hacer**
        $\langle t_{colision}, contactos\rangle := CalcularTiempoColision(A, B, escena \rightarrow tiempo, t_{minimo})$

        si $t_{colision} \neq$ `NULL` **entonces**
            casos
                $t_{colision} > t_{minimo}$ → **nada**
                $t_{colision} = t_{minimo}$ → reunir *contactos* en una normal **n** y un punto P
                crear colision c a partir de:
                A, B, t_{minimo}, *contactos*, P, **n**
                añadir c a $listaColisiones$
                $t_{colision} < t_{minimo}$ → reunir *contactos* en una normal **n** y un punto P
                crear colision c a partir de:
                A, B, t_{minimo}, *contactos*, P, **n**
                vaciar $listaColisiones$
                añadir c a $listaColisiones$
                $t_{minimo} := t_{colision}$
            fcasos
        fsi
    fpara
    devolver $\langle listaColisiones, t_{minimo}\rangle$

Algoritmo 5.4: Método $BuscarColisionesMasCercanas$

Capítulo 6

Resting Contact

Decimos que un conjunto de sólidos rígidos están en *resting contact* **si en cada punto de contacto la velocidad relativa en esos puntos es** 0, bajo cierta tolerancia TOL (Figura 6.1).

6.1. Tratamiento del *resting contact*

Recordemos que en caso de colisión calculábamos un impulso $j\mathbf{n}(t)$, donde j era un escalar y $\mathbf{n}(t)$ era la normal de colisión en el instante t. Para el *resting contact* debemos calcular una fuerza $f_i n_i(t)$ en cada punto de contacto i , donde f_i es un escalar desconocido y $\mathbf{n}_i(t)$ es la normal en el punto de contacto (ver Figura 6.2). Nuestro objetivo es **determinar el valor de cada** f_i. Por otro lado, todas las f_i deben ser determinadas en el mismo tiempo, ya que la fuerza aplicada en el punto de contacto i-ésimo puede intervenir en el punto de contacto j-ésimo.

Debemos calcular las f_i de acuerdo a tres condiciones:

- **Evitar la interpenetración:** las fuerzas de contacto deben ser lo suficientemente fuertes para prevenir que los dos cuerpos en contacto que se están empujando interpenetren. Para cada punto de contacto i construimos una expresión $d_i(t)$, que mide la separación entre dos cuerpos alrededor del punto de contacto en el tiempo t. Un valor positivo indica que se ha deshecho el contacto y los cuerpos se han separado. Un valor negativo indica interpenetración. Puesto que los cuerpos están en contacto en el tiempo actual t_0, tendremos $d_i(t_0) = 0$ (salvo tolerancia TOL). Nuestro objetivo es asegurarse que las fuerzas de contacto mantienen $d_i(t) \geq 0$ para cada punto de contacto en un tiempo futuro $t > t_0$.

  Sean $p_a(t)$ y $p_b(t)$ los puntos de contacto implicados en el contacto i-ésimo entre

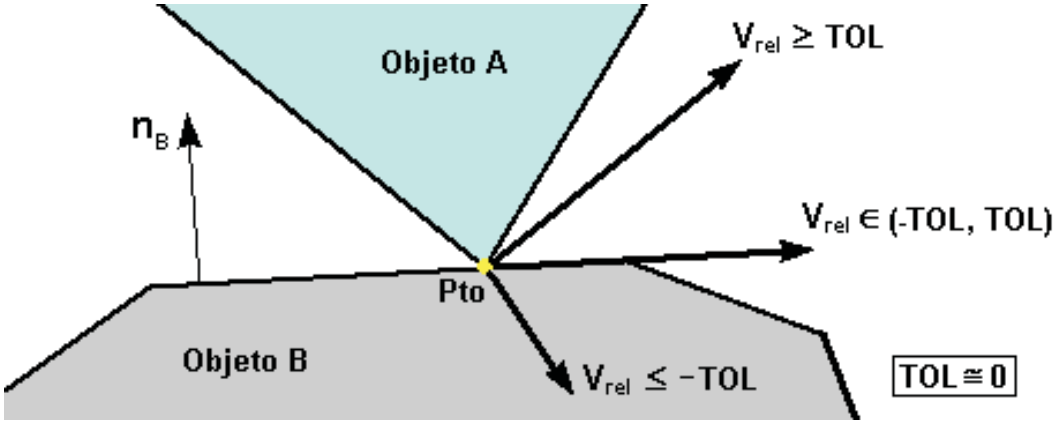


Figura 6.1: Dados dos objetos vemos el vector *velocidad relativa* para cada uno de los tres casos posibles, en función del valor de tolerancia TOL

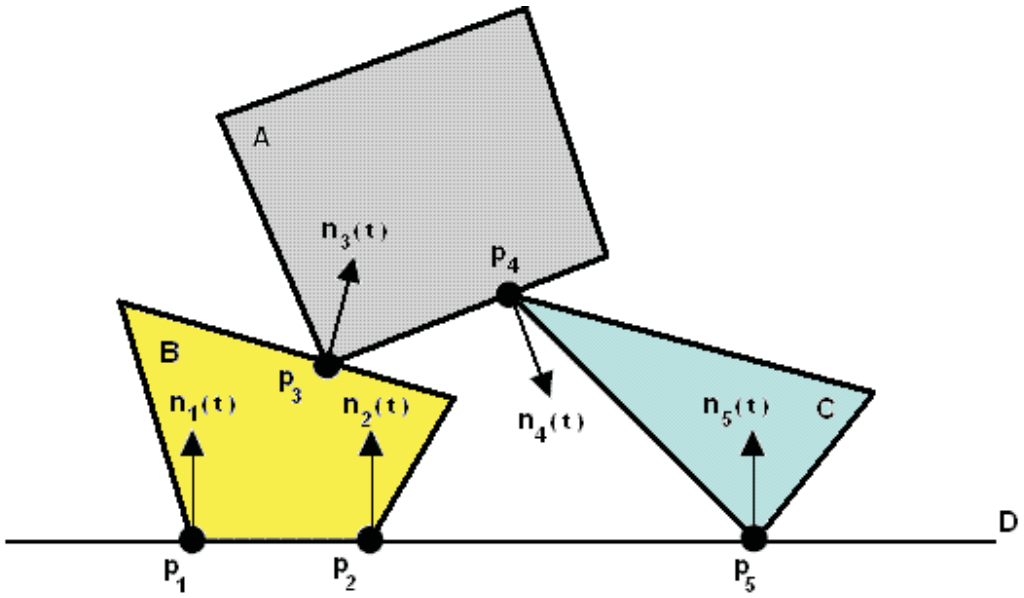


Figura 6.2: Objetos en *resting contact*. Se tienen 5 puntos de contacto. Una fuerza de contacto actúa entre pares de objetos en cada punto de contacto.

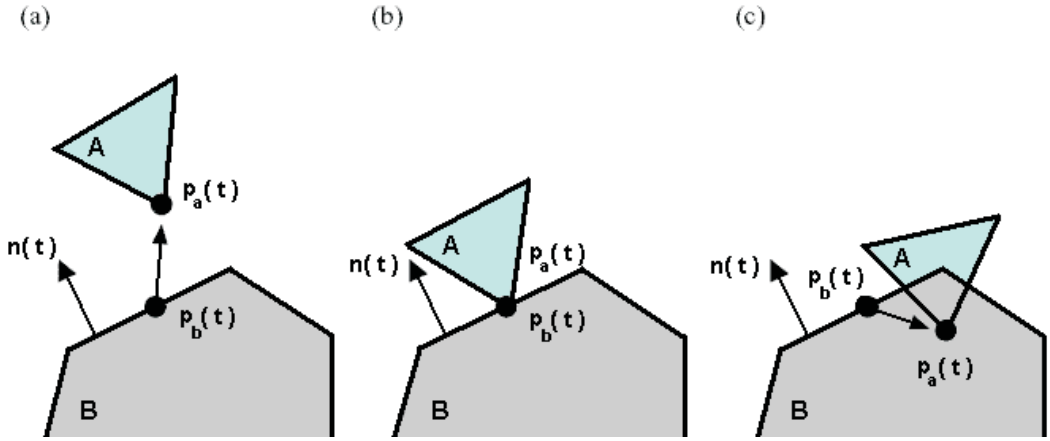


Figura 6.3: (a) El desplazamiento $p_a(t) - p_b(t)$, indicado por una flecha, apunta en el mismo sentido que $n(t)$. Por esto, la distancia $d(t)$ sería positiva. (b) La distancia $d(t)$ es 0. (c) El desplazamiento $p_a(t) - p_b(t)$ apunta en el sentido opuesto al de $n(t)$. En este caso, $d(t)$ es negativa, indicando interpenetración.

los cuerpos A y B, respectivamente (ver Figura 6.3). Definimos $d_i(t)$ de la siguiente forma:

$$d_i(t) = n_i(t) \cdot (p_a(t) - p_b(t))$$

Notación: Por motivos de simplicidad, en este apartado denotaremos la derivada primera de una función x respecto a t como $\dot{x}(t)$ y la derivada segunda como $\ddot{x}(t)$.

Como hemos dicho anteriormente, en el instante t tenemos que $d_i(t) = 0$. Tenemos que evitar que $d_i(t)$ decrezca, ya que se produciría interpenetración. Por lo tanto, tenemos que hacer que $\dot{d}_i(t) \geq 0$. ¿Qué indica $\dot{d}_i(t)$?. Puesto que $d_i(t)$ indica la distancia de separación, $\dot{d}_i(t)$ indicará la velocidad de separación en el instante t.

Por otra parte, derivando $d_i(t)$ se tiene que $\dot{d}_i(t) = n_i(t) \cdot (\dot{p_a}(t) - \dot{p_b}(t))$, siendo $\dot{p_a}(t)$ la velocidad puntual del objeto A en el punto de contacto entre ambos objetos y $\dot{p_b}(t)$ la velocidad puntual del objeto B. Es decir, tenemos que $\dot{d}_i(t) = n(t) \cdot (v_{pA}(t) - v_{pB}(t))$, lo cual es igual a la velocidad relativa calculada en la Ecuación (4.4). Cuando los cuerpos están en *resting contact*, $\dot{d}_i(t)$ es cero, porque ninguno de los cuerpos se están moviendo el uno hacia el otro sobre el punto de contacto. (La velocidad relativa es cero)

Resumiendo, si los cuerpos están en *resting contact*, $d_i(t) = 0$ y $\dot{d}_i(t) = 0$. ¿Qué sucede con $\ddot{d}_i(t)$?.

$\ddot{d}_i(t)$ mide cómo ambos objetos aceleran uno hacia el otro en el punto de contacto. Si $\ddot{d}_i(t) > 0$, los objetos tienen una aceleración que separa uno del otro, y la colisión se deshace inmediatamente después del instante t. Si $\ddot{d}_i(t) = 0$, continúa el contacto. El

caso $\ddot{d}_i(t) < 0$ debe ser evitado ya que esto indica que los puntos de contacto están acelerando el uno hacia el otro.

Por todo esto, la condición que debemos comprobar para **prevenir la interpenetración** es:

$$\ddot{d}_i(t) \geq 0 \tag{6.1}$$

que se debe de cumplir para cada punto de contacto.

- **La fuerza de contacto ha de ser repulsiva:** esto significa que cada f_i debe ser positiva, ya que una fuerza de $f_i\mathbf{n}_i(t)$ actúa sobre el cuerpo A, y $\mathbf{n}_i(t)$ es la normal dirigida hacia ese mismo cuerpo. Por lo tanto, necesitamos:

$$f_i \geq 0 \tag{6.2}$$

para cada punto de contacto.

- **Si los cuerpos comienzan a separarse la fuerza de contacto ha de ser 0**: esta restricción está expresada en términos de f_i y $\ddot{d}_i(t)$. Puesto que la fuerza de contacto f_i debe hacerse 0 si el contacto entre ambos cuerpos se deshace en el i-ésimo contacto, esto significa que f_i debe ser 0 en este caso. Esto quiere decir que:

$$f_i\ddot{d}_i(t) = 0 \tag{6.3}$$

Esto es así, ya que si tenemos que $\ddot{d}_i(t) > 0$, entonces para satisfacer la ecuación (6.3) se necesita que $f_i = 0$. Si el contacto entre ambos cuerpos no se está deshaciendo, ya que seguimos en *resting contact*, entonces $\ddot{d}_i(t) = 0$ y la ecuación (6.3) se satisface independientemente del valor de f_i.

Para encontrar las f_i que satisfacen las tres restricciones (6.1), (6.2) y (6.3), resulta que cada $\ddot{d}_i(t)$ puede expresarse en terminos de los f_i desconocidos como sigue [Bar97c]:

$$\ddot{d}_i(t) = a_{i1}f_1 + a_{i2}f_2 + ... + a_{in}f_n + b_i \tag{6.4}$$

Que expresado en forma matricial, resulta, para todo i:

$$\begin{pmatrix} \ddot{d}_1(t) \\ . \\ . \\ . \\ \ddot{d}_n(t) \end{pmatrix} = \mathbf{A} \begin{pmatrix} f_1 \\ . \\ . \\ . \\ f_n \end{pmatrix} + \begin{pmatrix} b_1 \\ . \\ . \\ . \\ b_n \end{pmatrix}$$

donde A es la matriz $n \times n$ de los coeficientes a_{ij} de la ecuación (6.4).

Los valores de a_{ij} y b_i con $1 \leq i, j \leq n$ son conocidos, por ello se trata de resolver un problema de la forma $D = Af + b$ sujeto a las condiciones (6.1), (6.2) y (6.3). En concreto, para resolver correctamente el problema del *resting contact* debemos calcular:

- La matriz A y el vector b (ver [Bar97c] para la derivación).
- A partir de ellos, el vector de fuerzas resultantes, es decir, las f_i. El sistema de ecuaciones formado por (6.1), (6.2), (6.3) y (6.4) da lugar a un problema de *programación cuadrática* (QP) [Gou00]. La programación cuadrática no es tan común como lo es la programación lineal. Los algoritmos que resuelven este tipo de problemas tienen una implementación más complicada, lo cual provoca en la mayoría de los casos el uso de paquetes comerciales que implementan resolutores.

  En [Bar94] se propone un algoritmo alternativo para el cálculo de fuerzas de contacto que no implica la resolución de un problema de programación cuadrática. En el caso de sistemas sin fricción el algoritmo es similar al algoritmo de Dantzig [CD68] para resolución de problemas de complementariedad lineal (LCP). Nuestro problema de *resting contact* adaptado a este algoritmo queda de la siguiente forma:

  Dados la matriz A y el vector $\mathbf{b}$, calculamos el vector $\mathbf{f}$ de fuerzas resultantes:

  - $A\mathbf{f} + \mathbf{b} \geq \mathbf{0}$, para evitar la interpenetración.
  - $f_i \geq 0, \forall i \in \{1..n\}$, para indicar que las fuerzas son repulsivas.
  - $\mathbf{f}^T(A\mathbf{f} + \mathbf{b}) = 0$, si los cuerpos comienzan a separarse.

  Este algoritmo alternativo fue el implementado para el cálculo de las fuerzas de contacto. Se describirá brevemente en la siguiente sección.

6.2. Resolutor LCP adaptado al problema de *resting contact*

El algoritmo a implementar funciona de la siguiente forma:

- Inicialmente todos los f_i toman valor 0.
- Buscamos un valor para f_1 que satisfaga las tres condiciones comentadas anteriormente.
- Buscamos un valor para f_2 que satisfaga las tres condiciones, manteniendo las condiciones para f_1.
- Buscamos un valor para f_3 que satisfaga las tres condiciones, manteniendo las condiciones para f_2 y f_1.
- etc.

Una descripción completa de dicho algoritmo y su demostración de corrección y completitud puede obtenerse en [Bar94]. Para el cálculo del incremento de los f_i que mantienen las tres condiciones en los f_k $(1 \leq k \leq i)$ es necesaria la resolución de un sistema de ecuaciones lineal. En *RBS* se ha utilizado el método de resolución mediante descomposición LU, cuya implementación fue tomada de [PTVF92].

6.3. Integración en el bucle de simulación

En la sección 5.3 se describía un bucle de simulación que consistía en la búsqueda de colisiones que se producen en el tiempo más cercano, tras lo cual se realiza su tratamiento (aplicación de impulso). A continuación trataremos de integrar el cálculo y aplicación de fuerzas de contacto en dicho bucle.

El Algoritmo 6.1 muestra un esquema del método $AvanzarHastaColisiones$ modificado. A continuación se detallan las diferencias más relevantes:

- **Inclusión de un conjunto** $restingContact$, donde se almacenan todos los contactos que son *resting contact*, es decir, en aquellos donde la velocidad relativa de los puntos de contacto es 0.
- **Cálculo de fuerzas de contacto**. Una vez que hemos tratado todas las colisiones, tenemos en el conjunto *restingContact* los contactos sobre los que deben aplicarse las fuerzas de contacto. Para hallar dichas fuerzas tenemos que calcular previamente la matriz A y el vector $\mathbf{b}$ de la adaptación del LCP comentado anteriormente, para después resolverlo mediante la llamada a $ResolverLCP$.
- **Aplicación de las fuerzas de contacto.** El resolutor LCP nos devuelve los módulos de las fuerzas que se deben aplicar en cada contacto. Sólo nos queda aplicar cada fuerza a su contacto correspondiente. Dicha fuerza tiene como módulo el valor de f_i calculado y como dirección la normal de contacto.

6.4. Problemas encontrados con *resting contact*

La implementación del tratamiento del *resting contact* en *RBS* no llegó a funcionar correctamente salvo para casos muy sencillos (esfera reposando sobre un paralelepípedo, esfera deslizándose por una rampa, etc.). Los principales problemas encontrados fueron:

- **Valores de salida del resolutor LCP incorrectos**: En [Bar94] se comenta que la corrección y completitud del algoritmo resolutor LCP está sujeta a una condición:

---

rbsEscena :: $AvanzarHastaColision(\Delta t)$
apilar el estado de todos los objetos de la escena
avanzar el estado de todos los objetos de la escena un tiempo Δt
$paresBB := SolapamientosPorBoundingBox()$
desapilar el estado de todos los objetos de la escena

$\langle colisiones, t_{minimo} \rangle := BuscarColisionesMasCercanas(paresBB, \Delta t)$

avanzar todos los objetos de la escena un tiempo t_{minimo}

repetir
restingContact := { }
hayColisiones := `FALSE`
para cada colision c de *colisiones* **hacer**
si los objetos involucrados en c se están acercando **entonces**
calcular y aplicar impulso a los objetos participantes en c
hayColisiones := `TRUE`
sino si los objetos involucrados en c están en *resting contact* **entonces**
añadir los contactos de c al conjunto de *restingContact*
fsi
fpara
hasta que *hayColisiones* = `FALSE`

si el conjunto *restingContact* no está vacío **entonces**
$A := CalcularMatrizA(restingContact)$
$\mathbf{b} := CalcularVectorB(restingContact)$
$\mathbf{f} := ResolverLCP(A, B)$
aplicar $\mathbf{f}$ a los objetos implicados en los contactos de *restingContact*
fsi
devolver t_{minimo}

Algoritmo 6.1: Método *AvanzarHastaColisiones* con resting contact

La matriz de entrada A debe ser **semidefinida positiva** (PSD). Una matriz M es semidefinida positiva si y sólo si $\mathbf{v}^T M\mathbf{v} \geq 0$ para todo $\mathbf{v} \neq \mathbf{0}$. En las pruebas que hicimos sólo conseguimos obtener una matriz semidefinida positiva en casos donde sólo había un contacto. En este caso se calculaba una matriz 1x1 cuyo único elemento era positivo (por tanto, la matriz era semidefinida positiva).

Sin embargo, en otras pruebas donde intervenían más contactos (como por ejemplo, un paralelepípedo sobre otro) la matriz A de entrada no era PSD. En estos casos se devolvían soluciones exageradamente grandes para los f_i.

- **No terminación del resolutor LCP**: Si la matriz de entrada A no es PSD, también es posible que el algoritmo no termine. Esto provocó que en algunos casos de prueba la simulación quedase "colgada".

- **Inestabilidad numérica**: En casos en que la salida del resolutor LCP se podía considerar correcta, ya que cumplía las condiciones expuestas al final de la sección 6.1, la fuerza de contacto realmente aplicada era ligeramente inferior a la que realmente se debía aplicar para evitar la interpenetración, debido principalmente a errores de redondeo. Esto provocaba que los objetos que estaban en *resting contact* interpenetrasen ligeramente. El tratamiento de las colisiones en nuestro bucle de simulación tiene en todo momento como finalidad evitar la interpenetración. Sin embargo, cuando los objetos se encuentran *ya* interpenetrando (bien sea porque el usuario los haya colocado así, bien sea por el error de redondeo que estamos comentando) se ignora dicha interpenetración, ya que en principio sería complicado averiguar qué impulso o fuerza de contacto se debería aplicar para separar los objetos. Por este motivo una pequeña interpenetración entre dos objetos por errores de redondeo tiene como consecuencia que en los siguientes ciclos de simulación no se detecte ni colisión ni resting contact entre los mismos, por lo que en los siguientes pasos la interpenetración será aún mayor.

Capítulo 7

Conclusiones

A lo largo de nuestro proyecto hemos llevado a cabo el estudio de diversos conceptos procedentes de la mecánica, análisis numérico y geometría computacional, hasta aplicarlos en la implementación de una API para la simulación de sólidos rígidos.

En una primera aproximación hemos tratado los sólidos rígidos desde el punto de vista de la cinemática y dinámica aplicadas a este tipo de cuerpos. Esto tuvo como resultado una primera versión de *RBS* que permitía simular el comportamiento de sólidos rígidos independientes.

Posteriormente se incorporaron a nuestro proyecto conceptos relacionados con la geometría computacional, que dieron como resultado la implementación de los diversos algoritmos de detección de colisiones y su integración en la arquitectura ya existente. Por otro lado siguió evolucionando el tratamiento desde el punto de vista físico de los sólidos rígidos, lo cual permitió incorporar el concepto de impulso. En este punto del proyecto ya estuvimos en disposición de realizar simulaciones con varios cuerpos interactuando unos con otros.

La última fase de nuestro proyecto consistió en obtener la implementación de una respuesta física realista al fenómeno conocido como *resting contact*. Esto dió de nuevo lugar a la presentación de nuevos conceptos físicos cuya implementación fue más compleja de lo inicialmente estimado. El resultado fue una implementación parcial del *resting contact*, que sólo llegó a funcionar en casos muy sencillos.

Paralelamente a todo este proceso de desarrollo de la API, se diseñó e implementó un modelo destinado a la representación de las escenas simuladas, siguiendo los pasos de la arquitectura modelo-vista-controlador (MVC). El producto de todo esto fue una interfaz completamente modular que no sólo permitía la visualización gráfica de la simulación desde varios puntos de vista, sino también la modificación de los parámetros que intervenían en la escena simulada (fuerzas, masas, comportamientos, etc.)

El resultado global del proyecto es una API potente y extensible para la simulación

en tiempo real de sólidos rígidos. Se ha buscado en todo momento un compromiso entre realismo y tiempo de cómputo.

7.1. Futuras mejoras/extensiones

7.1.1. Implementación estable de *resting contact*

En la sección 6.4 se comentó que el *resting contact* sólo llegó a funcionar en casos sencillos. Hay varias alternativas a la propuesta de [Bar97c], que fue la implementada en *RBS*. Un ejemplo es la simulación dinámica basada en el impulso propuesta en [Mir96b]. Esta última realiza un tratamiento de *resting contact* que no requiere el uso de ningún resolutor LCP.

7.1.2. Dinámica con restricciones

La idea de este tipo de dinámica es la incorporación de restricciones en el modo en el que las partículas se pueden mover. Por ejemplo, podemos obligar que un objeto se mueva a lo largo de una curva previamente definida o imponer que dos partículas permanezcan separadas en todo momento una determinada distancia.

El problema de la dinámica con restricciones consiste en buscar que los objetos se comporten de acuerdo a las leyes de Newton y de acuerdo a ciertas restricciones geométricas. Su implementación supondría la posibilidad de poder simular cuerpos articulados o unidos entre sí. Podemos encontrar una introducción a este problema en [Bar97a].

7.1.3. Tamaño de paso adaptable

Uno de los problemas a la hora de simular es la elección de un tamaño de paso adecuado. Tamaños de paso demasiado grandes conducen a un aumento del error cometido. Tamaños de paso demasiado pequeños producen cálculos innecesarios e inestabilidad numérica.

Una posible solución a esto pasa por la implementación de un tamaño de paso que varíe con el tiempo. Por ejemplo, a partir de un determinado estado de la escena podemos calcular dos estimaciones para el estado siguiente: una realizando un paso de un determinado tamaño y otra realizando dos pasos de la mitad de este tamaño. Si la diferencia entre estas dos estimaciones es muy grande, debemos reducir el tamaño de paso.

7.1.4. Tratamiento de eventos

De momento los únicos eventos tratados en *RBS* son los notificados a los observadores de la escena cuando se ha realizado un paso de simulación. Si se quiere aplicar este proyecto a la programación de videojuegos, el usuario de la API querrá conocer en qué momento se produce una determinada colisión o cuándo se detiene un determinado objeto. Este tipo de eventos deben detectarse y notificarse.

7.1.5. Otras mejoras

- **Ampliación del número de geometrías contempladas.**
- **Ampliación del número de algoritmos de detección de colisiones.**
- **Incorporación de fricción estática/dinámica.**
- **Incorporación de nuevos tipos de fuerzas (muelles, viento, etc.)**
- etc.

Apéndice A

Interfaz de usuario y capturas de escenas

A.1. Interfaz de usuario

En este proyecto, además de haber implementado el API *RBS*, hemos desarrollado una interfaz interactiva con el usuario que permite visualizar y modifcar datos relativos a los objetos y a las fuerzas que actúan sobre ellos.

La ventana principal de nuestra interfaz tendrá el aspecto mostrado en la figura A.1. En este ejemplo se tienen dos objetos, una esfera y un paralelepípedo.

Disponemos de tres vistas distintas para cada escena. En la figura A.1, la vista de la izquierda es la frontal, la del centro es la cenital y la de la derecha es la lateral.

Los datos que se pueden visualizar sobre ambos objetos son: el nombre que reciben, su masa, qué comportamientos (lineal y rotacional) tienen activos/inactivos, el ángulo que forman con respecto a cada eje coordenado y la posición de su centro de masas.

Nuestra interfaz también permite modificar ciertas propiedades de los objetos de la escena que tienen un nombre asignado. Esas propiedades son la masa y sus comportamientos (lineal y rotacional).

Además, podemos avanzar indefinidamente la simulación mediante el botón *Avanzar* o avanzar un único paso en la simulación mediante el botón *Avanzar paso*.

También se muestra el tiempo actual de simulación.

Al pulsar el botón *Gestor de fuerzas* se desplegará la ventana mostrada en la figura A.2. En una tabla se muestran todas las fuerzas que actúan, en el paso de simulación actual, sobre cada objeto con nombre que hay en la escena. Los datos mostrados son: el nombre del objeto al que afecta la fuerza, el tipo de fuerza (instantánea, temporal, perpetua, ...),

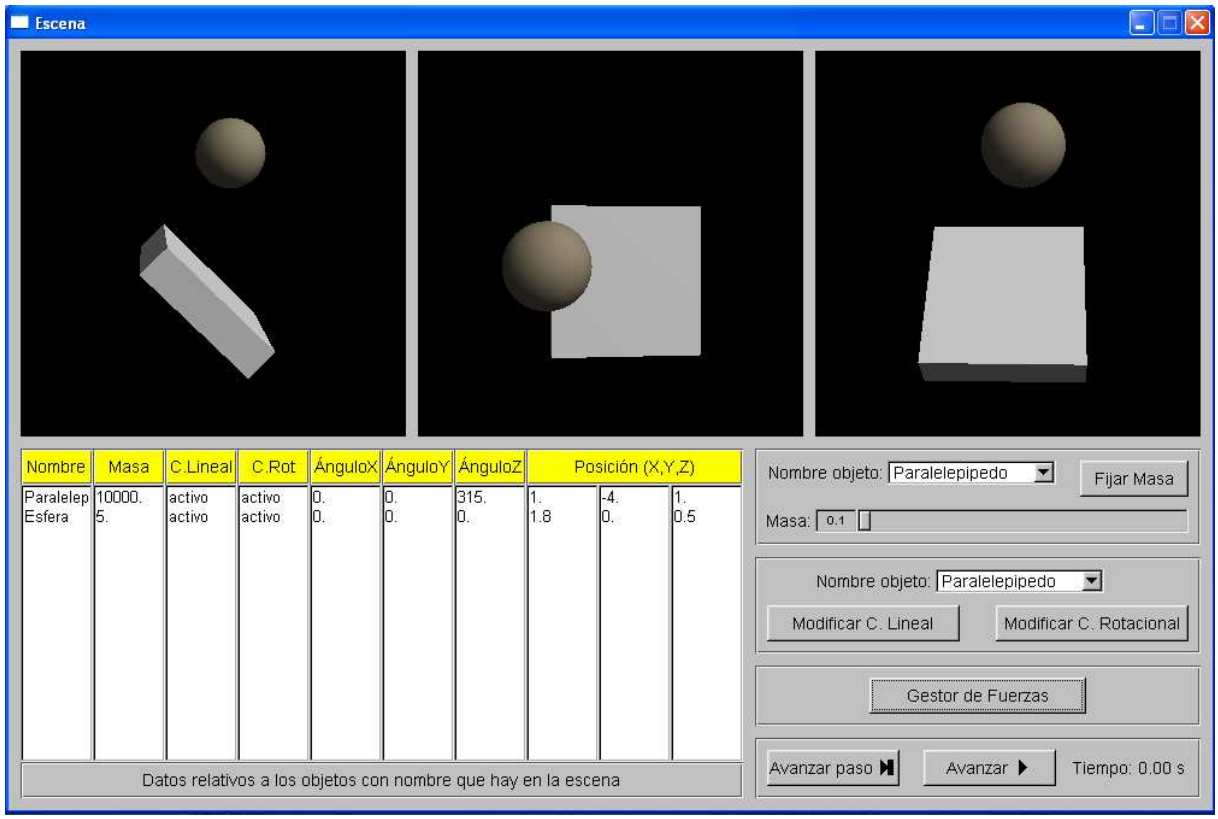


Figura A.1: Ventana principal de nuestra interfaz.

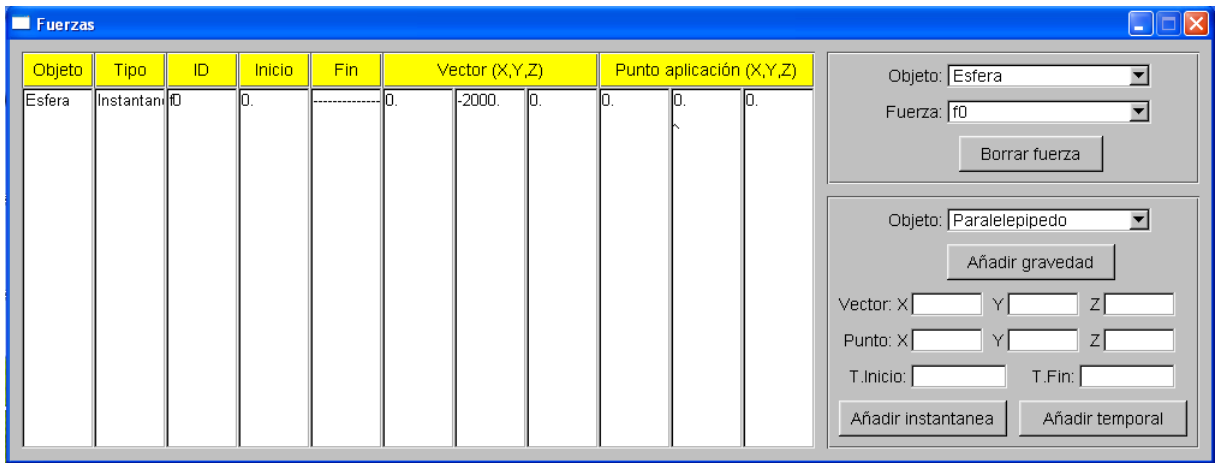


Figura A.2: Ventana para la getión de fuerzas de los objetos con nombre de la escena.

un identificador único de fuerza generado autómáticamente por la interfaz, los instantes de inicio y el de finalización de la fuerza, el vector que representa a la fuerza y su punto de aplicación sobre el objeto. Por otra parte, se podrán borrar fuerzas seleccionando el objeto al que pertenence y su identificador. También se permite añadir fuerzas a los objetos de tres tipos: gravedad, instantánea y temporal. Para ello, debemos indicar el objeto al cual se quiere añadir la fuerza y en función del tipo de fuerza una serie de datos concretos. De esta forma para añadir gravedad no necesitamos ningún dato adicional; para añadir una fuerza instantánea necesitamos el vector de la fuerza, su punto de aplicación y el instante de comienzo; y para añadir una temporal se necesita el vector de la fuerza, su punto de aplicación y los instantes de comienzo y finalización.

Para la implementación de la interfaz de usuario, hemos utilizado la librería FLTK (Fast Light Toolkit). Esta librería maneja componentes de interfaces de usuario (ventanas, botones, menús, etc.). Es compatible con OpenGL, y funciona en Windows, UNIX/Linux y MacOS.

Hemos empleado el modelo *vista-controlador*. De esta forma, cada elemento de la interfaz será tratado como una componente. Por ejemplo, a la hora de modificar la masa de un objeto (ver figura A.1), hemos implementado la clase **rbsControladorMasa**, que dispone de un *choice* para seleccionar el nombre del objeto al que se le quiere modificar la masa, de una barra deslizadora para introducir el valor de la masa y de un botón para fijar los cambios. Todo ello es tratado como una única componente en la interfaz.

La clase **rbsVistaControlador** contiene instancias de cada una de estas componentes y es la encargada de situarlas en la interfaz principal.

A.2. Capturas de escenas

A continuación mostramos algunas de las escenas que hemos implementado. En ellas se puede observar la evolución de los objetos a lo largo del tiempo. Para cada escena mostraremos las tres vistas que permite nuestra interfaz.

A.2.1. Escena 1

En esta escena se muestra algo semejante a un billar. A la bola blanca se le aplica una fuerza instantánea sobre su centro de masas, en el instante $t = 0$ y cuyo vector es el $(5000, 0, 0)$.

La bola blanca colisionarán con el resto de bolas en el instante $t = 1{,}06$. Nos interesa ver las colisiones que se producen entre las esferas y con la mesa de billar (construída mediante paralelepípedos). La figura A.3 representa la escena en tres instantes diferentes de tiempo.

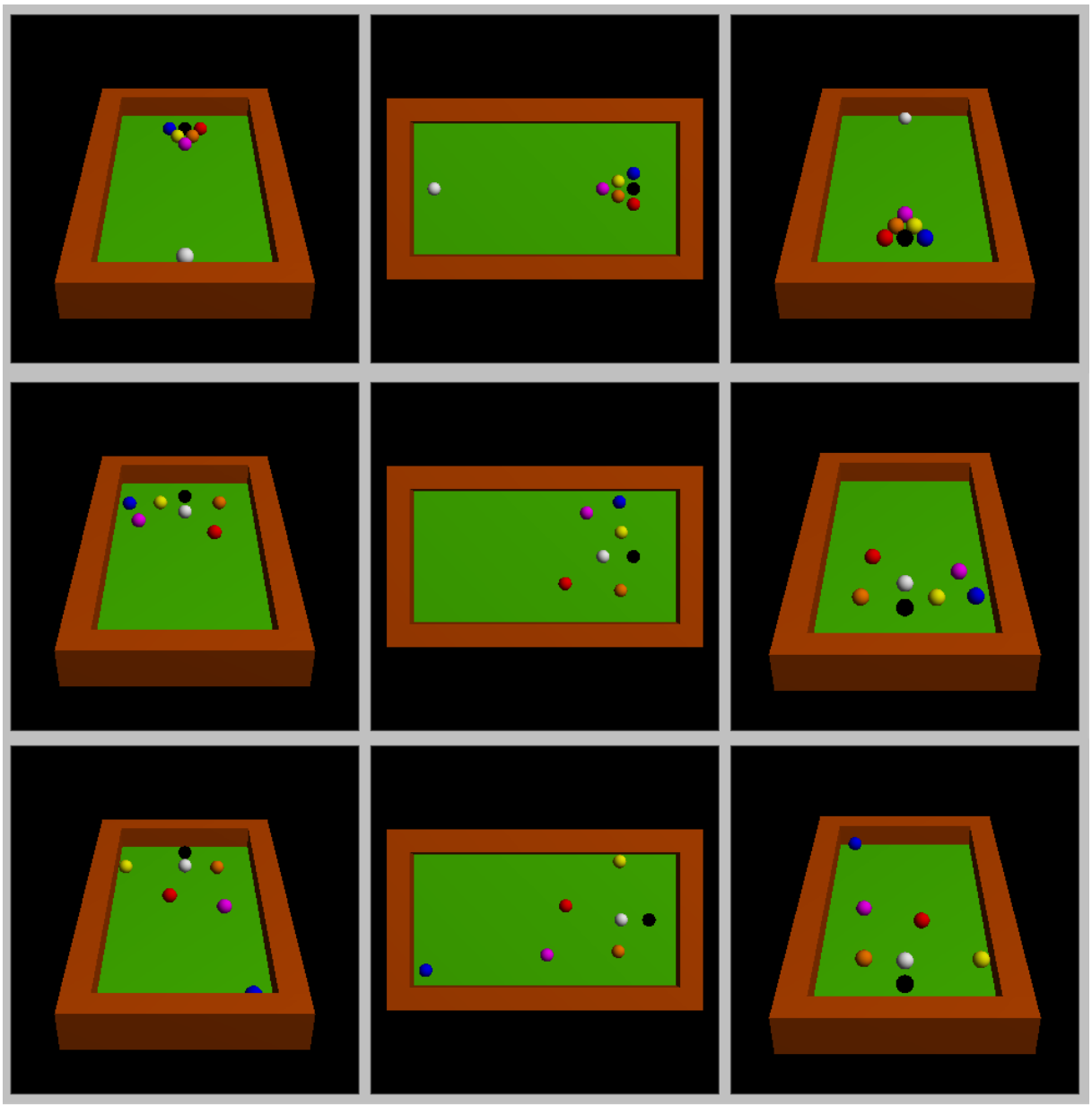


Figura A.3: La primera fila muestra diferentes vistas de la escena en el instante $t = 0$. La segunda fila muestra la escena en $t = 2{,}92$ segundos. La tercera fila sucede en el instante $t = 7{,}46$ segundos.

A.2.2. Escena 2

En esta escena mostramos una serie de paralelepípedos. La ficha amarilla está sometida a la fuerza de la gravedad. Por lo tanto, esta ficha caerá sobre las de abajo.

Vemos la figura A.4. Las dos fichas de la izquierda tienen tanto su comportamiento lineal como el rotacional desactivados, y por lo tanto, permanecerán en su posición pase lo que pase, es decir, ni rotarán ni se trasladarán. La ficha amarilla tiene ambos comportamientos activos. El resto de las fichas tienen el comportamiento rotacional activado y el lineal desactivado, y por esto, simplemente rotarán.

A.2.3. Escena 3

Nuestro objetivo en esta escena (ver figura A.5) es el de mostrar la potencia de nuestro API en el caso de múltiples colisiones simultáneas. Tenemos 30 objetos. En el instante $t = 0$, a algunos de ellos se les aplica una fuerza instantánea. En el instante $t = 0{,}3$, se les aplica otra fuerza instantánea distinta a los dos paralelepípedos frontales (los amarillos), de forma que éstos colisionen posteriormente con el resto de objetos. Las seis esferas blancas superiores están afectadas por la gravedad desde el instante inicial de la simulación ($t = 0$).

A.2.4. Escena 4

En esta escena (ver figura A.6) mostramos el caso del *resting contact*. Inicialmente la esfera está suspendida sobre la rampa. Como está afectada por la gravedad caerá hacia la rampa. Por otra parte, el coeficiente de restitución entre la esfera y la rampa es 0. Esto hará que cuando ambos objetos entren en contacto, la velocidad relativa en el punto de contacto sea prácticamente 0 y, por lo tanto, se produce *resting contact*. En la segunda fila de la figura A.6 se muestra este instante. Gracias a esto, la pelota no atraviesa al paralelepípedo, sino que se traslada sobre él (como muestra la tercera fila de la figura A.6).

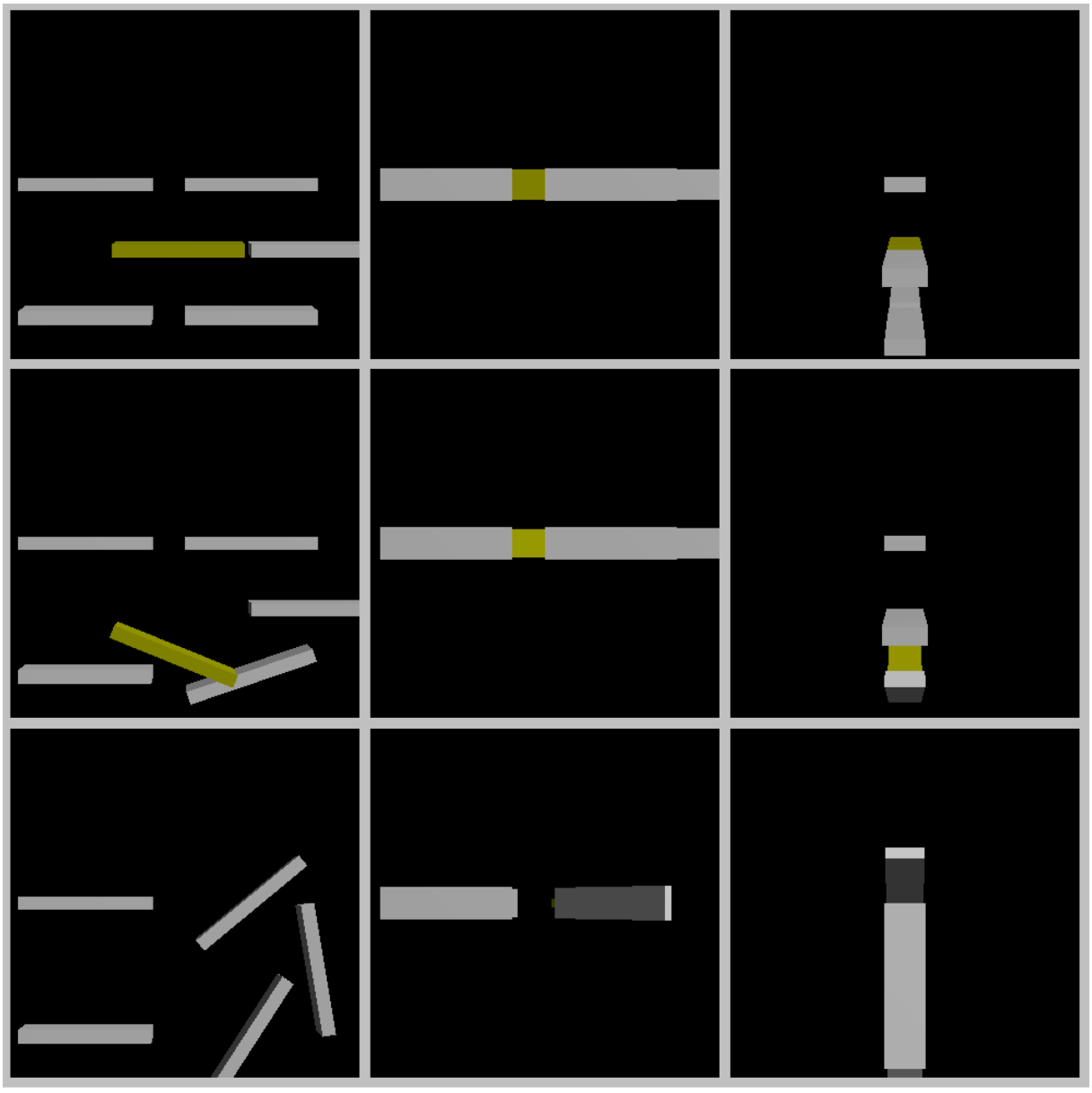


Figura A.4: La primera fila muestra diferentes vistas de la escena en el instante $t = 0$. La segunda fila muestra la escena en $t = 0{,}52$ segundos. La tercera fila sucede en el instante $t = 2{,}68$ segundos.

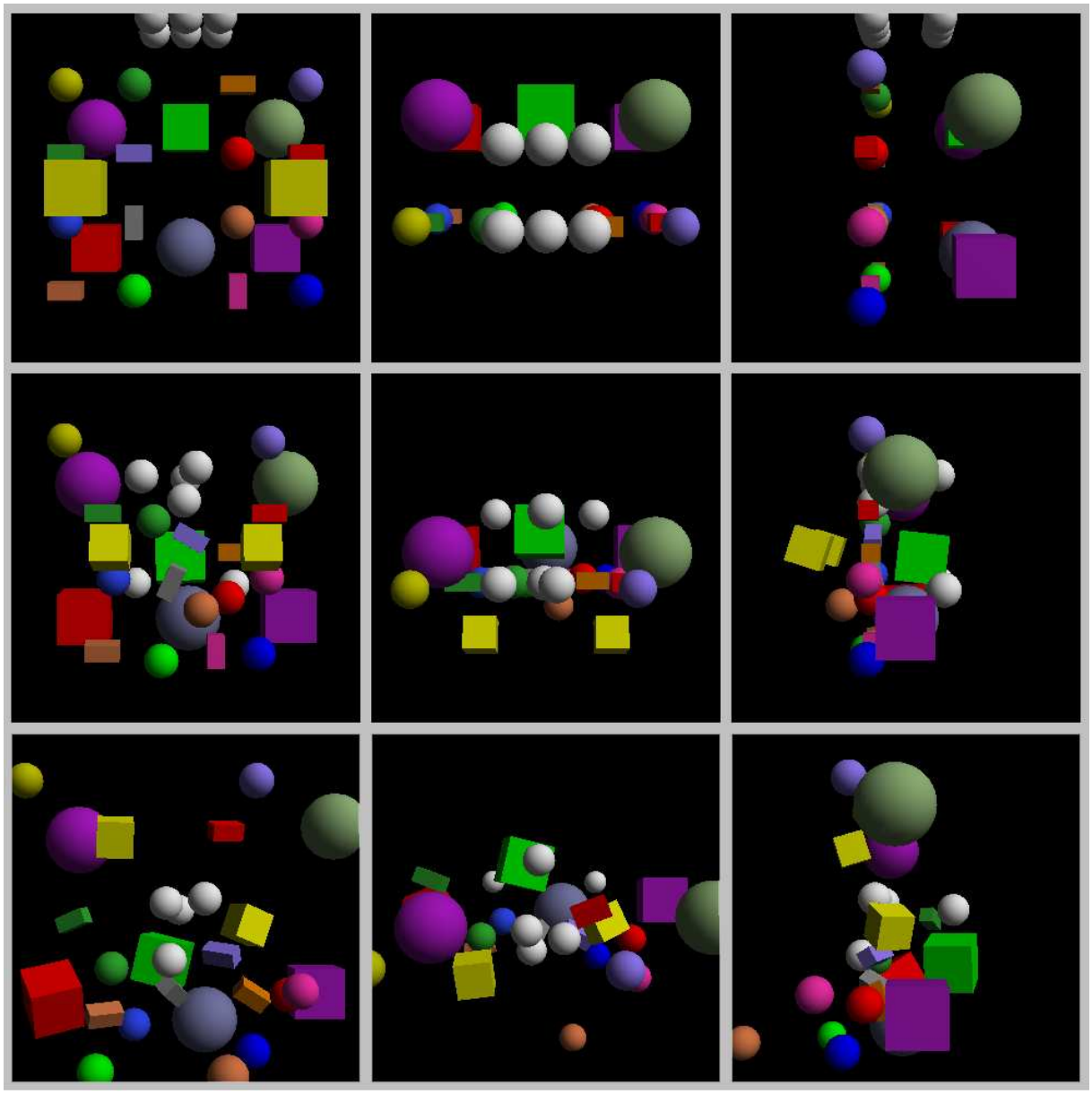


Figura A.5: La primera fila muestra diferentes vistas de la escena en el instante $t = 0$. La segunda fila muestra la escena en $t = 1{,}05$ segundos. La tercera fila sucede en el instante $t = 1{,}80$ segundos.

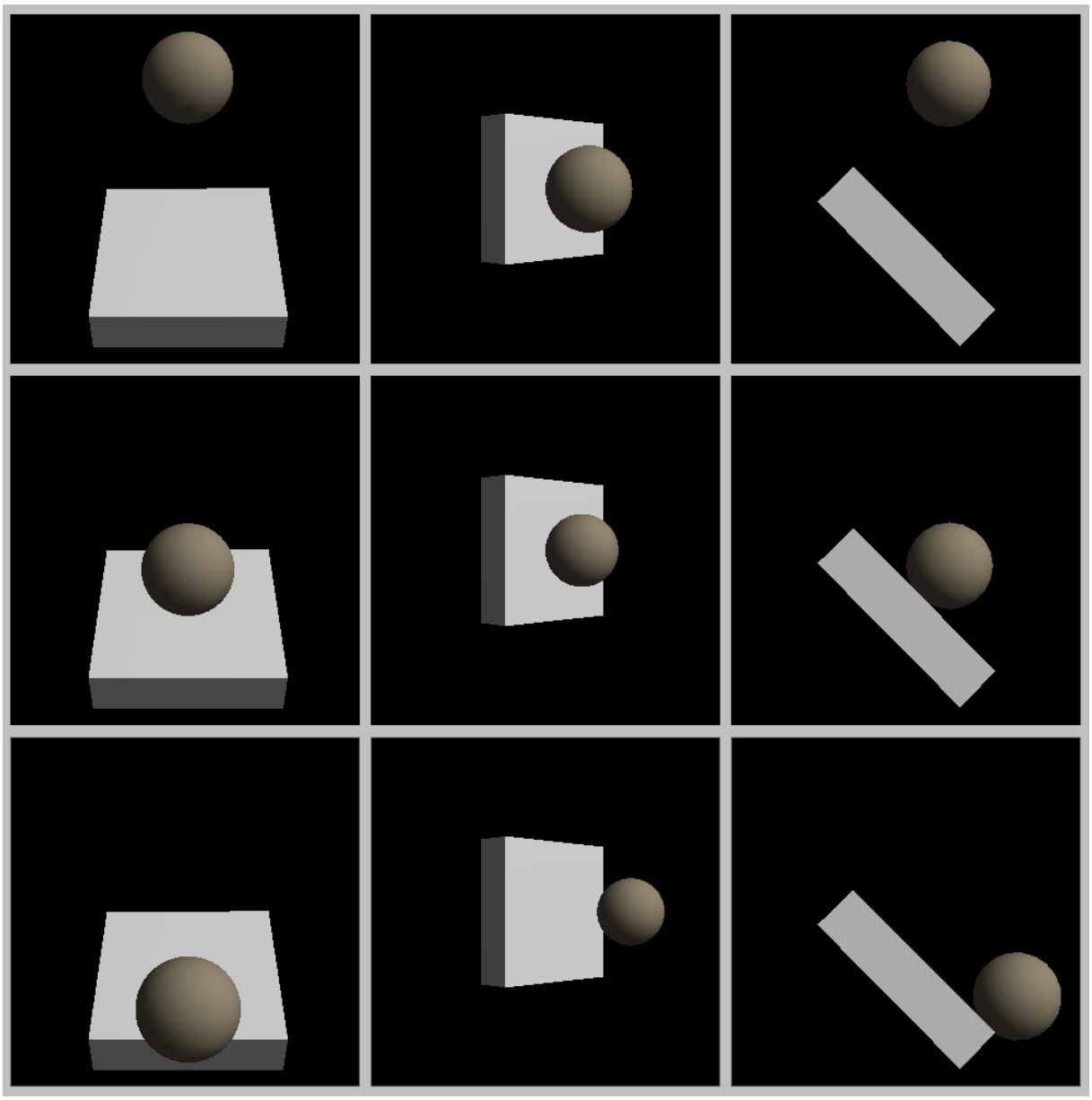


Figura A.6: La primera fila muestra diferentes vistas de la escena en el instante $t = 0$. La segunda fila muestra la escena en $t = 0{,}76$ segundos, justo cuando se produce *resting contact*. La tercera fila sucede en el instante $t = 1{,}10$ segundos.

Bibliografía

[Bar89] David Baraff. Analytical methods for dynamic simulation of non-penetrating rigid bodies. In *SIGGRAPH '89. Computer Graphics Proceedings*, volume 23, July 1989.

[Bar94] David Baraff. Fast contact force computation for nonpenetrating rigid bodies. In *SIGGRAPH 94, Computer Graphics Proceedings, Annual Conference Series*, pages 24–34, July 1994.

[Bar97a] David Baraff. Constrained dynamics. In *An Introduction to Physically Based Modelling. SIGGRAPH '97 course notes.*, 1997.

[Bar97b] David Baraff. Differential equation basics. In *An Introduction to Physically Based Modelling. SIGGRAPH '97 course notes.*, 1997.

[Bar97c] David Baraff. Rigid body simulation. In *An Introduction to Physically Based Modelling. SIGGRAPH '97 course notes.*, 1997.

[BF98] Richard L. Burden and J. Douglas Faires. *Análisis Numérico*, chapter 5.4, pages 276–286. International Thomson Ed., sixth edition, 1998.

[Bou01] David M. Bourg. *Physics for Game Developers.* O'Reilly, first edition, November 2001.

[CD68] Richard W. Cottle and George B. Dantzig. Complementary pivot theory of mathematical programming. In *Linear Algebra and its Applications*, pages 103–125, 1968.

[Cur02] Pablo Andrés Curello. Simulación de la dinámica de cuerpos rígidos en tiempo real. Instituto Tecnológico de Buenos Aires, 2002.

[Ehm99] Stephen Ehmann. Rigid body simulation tutorial. 1999.

[GBF03] Eran Guendelman, Robert Bridson, and Ronald Fedkiw. Nonconvex rigid body with stacking. SIGGRAPH 2003, 2003.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Patrones de Diseño. Elementos de software orientado a objetos reutilizable*, chapter 5, pages 289–298. Addison-Wesley, first edition, 1995.

[GJK88] Elmer G. Gilbert, Daniel W. Johnson, and S. Sathija Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. In *IEEE Journal of Robotics and Automation*, 1988.

[Gou00] Nick Gould. Quadratic programming: Theory and methods. In *3rd FNRS Cycle in Mathematical Programming*, February 2000.

[Hec96] Chris Hecker. Physics, Part 1: The next frontier. *Game Developer Magazine*, pages 12–20, October/November 1996.

[Hec97a] Chris Hecker. Physics, Part 2: Angular effects. *Game Developer Magazine*, pages 14–22, December/January 1996-1997.

[Hec97b] Chris Hecker. Physics, Part 3: Collision response. *Game Developer Magazine*, pages 11–18, March 1997.

[Hec97c] Chris Hecker. Physics, Part 4: The third dimension. *Game Developer Magazine*, pages 15–26, June 1997.

[Kap91] Wilfred Kaplan. *Advanced Calculus*, chapter 5, pages 285–291. Addison-Wesley, 4th edition, 1991.

[KEP05] Danny M. Kaufman, Timothy Edmunds, and Dinesh K. Pai. Fast frictional dynamics for rigid bodies. In *ACM Transactions on Graphics (SIGGRAPH 2005)*, August 2005.

[Lin93] Ming Chieh Lin. *Efficient Collision Detection for Animation and Robotics*. PhD thesis, University of California at Berkeley, December 1993.

[Mir96a] Brian Vincent Mirtich. Fast and accurate computation of polyhedral mass properties. *Journal of Graphics Tools*, 1(2), 1996.

[Mir96b] Brian Vincent Mirtich. *Impulse-based Dynamic Simulation of Rigid Body Systems*. PhD thesis, University of California at Berkeley, 1996.

[Mir97] Brian Vincent Mirtich. V-Clip: Fast and robust polyhedral collision detection. Technical report, MERL - Mitshubishi Electric Research Laboratory, June 1997.

[Mir98] Brian Vincent Mirtich. Rigid body contact: Collision detection to force computation. Technical report, MERL - Mitshubishi Electric Research Laboratory, March 1998.

[Mir00] Brian Vincent Mirtich. Timewarp rigid body simulation. In *SIGGRAPH 00 Conference Proceedings*, July 2000.

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C*, chapter 2, pages 43–50. Cambrigde University Press, 2nd edition, 1992.