

# OpenSIM

Juan Coll Soler

# Índice General

<b>1. Resumen</b>	<b>4</b>
<b>2. Situación actual</b>	<b>5</b>
<b>3. Teoría computacional de las partículas</b>	<b>7</b>
3.1. Introducción a la teoría de partículas . . . . .	7
3.2. Estructura de un sistema de partículas . . . . .	9
3.3. Métodos de integración ("Solvers" numéricos) . . . . .	13
3.3.1. Introducción a los métodos de integración . . . . .	13
3.3.2. Serie de Taylor . . . . .	15
3.3.3. Euler . . . . .	17
3.3.4. Runge-Kutta 2 . . . . .	19
3.3.5. Runge-Kutta 4 . . . . .	23
3.3.6. Otros métodos . . . . .	27
3.4. Fuerzas . . . . .	28
3.4.1. Viscosidad . . . . .	28
3.4.2. Gravitacional . . . . .	28
3.4.3. Eléctrica . . . . .	29
3.4.4. Elástica amortiguada . . . . .	29
3.5. Funciones de energía . . . . .	31
3.6. Colisiones . . . . .	35
3.6.1. Colisión con un Plano . . . . .	35
3.6.1.1. Condición de existencia de una colisión . . . . .	35
3.6.1.2. Como encontrar el punto de colisión . . . . .	36
3.6.1.3. Colisión elástica pura . . . . .	37
3.6.1.4. Colisión amortiguada en la normal al plano . . . . .	38
3.6.2. Colisión con un Triángulos . . . . .	39
3.6.2.1. Condición de existencia de una colisión . . . . .	39
3.6.2.2. Tratamiento de la colisión . . . . .	41
3.7. Estructura de voxels . . . . .	42
<b>4. OpenSIM – Librería C++</b>	<b>45</b>
4.1. Introducción a la librería OpenSIM . . . . .	45
4.1. Diseño de clases . . . . .	46

4.2	Nomenclatura . . . . .	50
4.3	Explicación de las clases . . . . .	xx
4.3.1	La classe madre SimObject . . . . .	xx
4.3.2	Primitivas . . . . .	xx
4.3.3	Estructuras secundarias . . . . .	xx
4.3.4	Estructuras de datos . . . . .	xx
4.3.5	Estructuras de calculo . . . . .	xx
4.3.6	Clases de entrada y salida a disco . . . . .	xx
4.3.7	Clases de Salida a Pantalla . . . . .	xx
4.4	Conclusiones sobre la librería . . . . .	xx
<b>5.</b>	<b>Ejemplos de simulación . . . . .</b>	<b>xx</b>
5.1.	Simulación de sistemas físicos básicos . . . . .	xx
5.1.1.	Comportamiento de electrones en un campo eléctrico .	xx
5.1.2.	Comportamiento de la gravedad entre planetas . . . .	xx
5.1.3.	Simulación de una cuerda elástica . . . . .	xx
5.2.	Modelado de sistemas más complejos . . . . .	xx
5.2.1.	Simulación de agua . . . . .	xx
5.2.2.	Simulación de gases . . . . .	xx
5.2.3.	Fuego – Fuegos artificiales . . . . .	xx
5.2.4.	Explosiones . . . . .	xx
5.2.5.	Simulación de ropa . . . . .	xx
5.2.6.	Simulación de grupos animales (añadiendo heurística de comportamiento) . . . . .	xx
5.3.	Reconstrucción de objetos 3D a partir de un campo de fuerzas . . . . .	xx
<b>6.</b>	<b>Resultados</b>	<b>xx</b>
<b>7.</b>	<b>Conclusiones</b>	<b>xx</b>
<b>8.</b>	<b>Líneas Futuras</b>	<b>xx</b>
<b>9.</b>	<b>Bibliografía</b>	<b>xx</b>

# 1. Resumen

La finalidad del proyecto es la programación de una librería C++ que permita la creación y simulación de sistemas de partículas sobre cualquier plataforma. La librería libera al programador de todo el trabajo de programación y optimización que requieren estos sistemas dejándolo concentrarse en su aplicación concreta.

Para entender y aprovechar las posibilidades de la librería explicaremos la teoría de los sistemas de partículas y mostraremos ejemplos de aplicaciones de estos sistemas de forma teórica por una parte y práctica por otra, utilizando la librería.

La librería no solo sirve para sistemas de partículas sino que se ha diseñado también para poder ampliar su utilización a muchos otros campos de la simulación. Quiere ser una base para la realización de proyectos futuros más específicos.

## 2. Situación actual

La simulación por ordenador siempre ha sido (desde que existe) una gran ayuda para la ingeniería. Hoy en día, con la potencia de cálculo existente se pueden simular los experimentos gráficamente. Por ejemplo podríamos ver el flujo del aire sobre las alas de un avión o, como se puede ver en la figura 2.1, la acción aerodinámica de la carrocería de un coche sobre el aire. Toda la simulación es virtual con las ventajas materiales que eso proporciona.

Básicamente tenemos 2 campos en la simulación:

1. Los objetos rígidos (una mesa, un coche, una rueda ...).
2. Los objetos deformables (el fuego, el agua , el gas, una pelota de goma, la ropa ....).

En el ejemplo de la figura 2.1 coexisten los dos campos, los rígidos (el coche) y los deformables (el aire). Para estos últimos se utilizan los sistemas de partículas, tema principal de este proyecto.

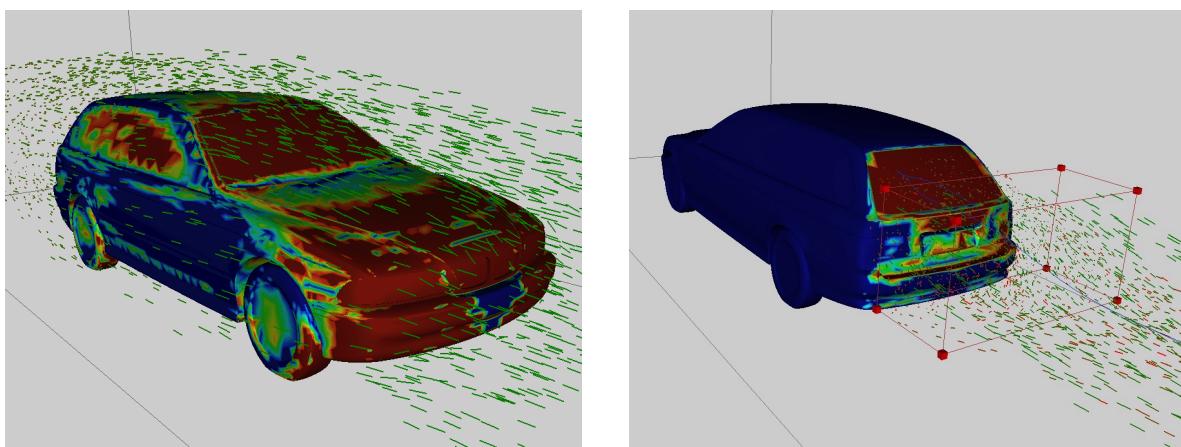


Figura 2.1 Imágenes obtenidas de :  
<http://www.vis.informatik.uni-stuttgart.de/~roettger/html/main/flow.html>

A continuación veremos las diferentes herramientas o programás que podemos encontrar hoy en día para solucionar la simulación de partículas.

Explicaremos también las razones por las cuales hemos preferido crear nuestra propia librería.

Actualmente se puede encontrar un gran número de programas que integran sistemas de partículas como por ejemplo el 3D Studio Max o el Macromedia Director. El problema de estas aplicaciones es la poca libertad que nos dejan, ya que tienen un lenguaje de programación restringido al entorno concreto de la aplicación.

Existen *motores 3D* (núcleo de programación) para juegos o aplicaciones que integran el tratamiento de partículas entre otras funciones. Karma de Mathengine es un ejemplo de motor 3D que tiene todas las características para simulaciones tanto científicas como lúdicas. El principal inconveniente de estos motores es su precio.

A nivel de usuario particular encontramos muchos códigos fuentes o pequeños motores que nos permiten crear partículas y simulaciones gráficas. Existen los famosos tutoriales de NEHE ([www.gamedev.net](http://www.gamedev.net)) por ejemplo. Son interesantes a título informativo pero no tienen la estructura de un librería general para la simulación.

Hemos preferido empezar de nuevo y no aprovechar lo que se encuentra, principalmente por 2 razones:

1. Ampliar nuestros conocimientos sobre los sistemas de partículas y la simulación gráfica en general. Esto nos permitirá la creación, modificación y adaptación de sistemas más específicos.
2. Crear una librería básica para todo tipo de aplicación en el ámbito de la simulación. Servirá de base para múltiples proyectos que irán aumentando la librería. De esta manera reaprovecharemos los códigos y evitaremos la redundancia de problemas básicos en gráficos.

# 3. Teoría computacional de partículas

## 3.1 Introducción

Una partícula es un objeto físico sin dimensión. Solo posee una posición en el espacio, una masa, y reacciona a fuerzas externas (o internas) al sistema. Es el objeto más fácil de simular por su simplicidad. Muchos fenómenos físicos se simulan con la ayuda de partículas o cogiendo el centro de gravedad de los objetos como único punto de interacción. Es el caso con fines gráficos, de sistemas gravitacionales como el sistema solar, o de sistemas eléctricos. Son sistemas donde no importa la rotación del objeto, solo su posición. La simulación de objetos rígidos no entra dentro de estos sistemas y tampoco en los objetivos de este proyecto.

La simplicidad de la partícula no implica la simplicidad de los comportamientos simulados. Uniéndolas con ciertas fuerzas, como por ejemplo las de un resorte, podremos simular sistemas elásticos complejos como la simulación de la ropa o de un gas. Simplemente encontrando las fuerzas pertinentes a nuestro sistema podremos simular un gran número de fenómenos complicados. Veremos más adelante ejemplos de la utilización de las partículas en este sentido. Uno de los más importantes es la reconstrucción de objetos 3D utilizando campos de fuerzas creados a partir de fotos o de escáneres. Su utilidad en medicina es considerable ya que permite pasar de radiografías (objeto 2D) a un objeto 3D observable y manipulable.

Esta simplicidad también resulta muy ventajosa para la simulación en tiempo real.

En las secciones siguientes veremos un poco de teoría sobre los sistemas de partículas. Explicaremos su estructura básica, cómo aplicarles las fuerzas y calcular sus nuevas posiciones. Veremos que necesitamos métodos de integración para el cálculo de las trayectorias (posición de las partículas) ya que utilizaremos las ecuaciones de la Mecánica de Newton para este fin. Existen diferentes métodos para dicha integración, y siempre deberemos jugar con el compromiso entre velocidad y precisión del sistema.

Haremos una lista de las fuerzas más utilizadas y explicaremos un método para encontrar fuerzas a partir de una condición determinada. Esto es útil cuando el sistema tiene que tender hacia una condición, por ejemplo en una simulación de un grupo de peces si queremos mantener los peces unidos, la condición límite podría ser que la posición del pez coincida con la del centro de masa del grupo.

Las colisiones son uno de los temas más complejos de la simulación. Muchos proyectos de investigación se dedican a este tema. En este proyecto sólo veremos las colisiones de partículas contra planos o triángulos. Son seguramente la más sencillas pero también son rápidas de calcular lo cual será útil en simulaciones en tiempo real. Con unos cuantos planos podremos simular colisiones bastante elaboradas.

Por último explicaremos las estructuras de voxels, que proporcionan una manera de discretizar el espacio 3D. La estructura de datos asociada requiere mucho espacio de memoria pero nos permite asignar a cada región del espacio una fuerza determinada y de rápido acceso. Esta estructura es muy útil cuando no se pueden representar fuerzas con una función matemática  $f(x, y, z)$  elemental o simple.

## 3.2 Estructura de un sistema de partículas

Consideraremos los sistemas de partículas como conjuntos de partículas de las que individualmente se conocen las fuerzas aplicadas, por lo tanto comenzaremos por estudiar el comportamiento de una sola partícula.

A una partícula en movimiento relativamente lento respecto de la velocidad de la luz (este es nuestro caso) se le pueden aplicar las leyes newtonianas de la mecánica. Lo que interesa conocer aquí es la trayectoria de la partícula a partir de la fuerza que se le aplica. A partir de ahora utilizaremos siempre la notación  $\mathbf{x} = (x, y, z)$  para indicar la posición de una partícula, y en general utilizaremos negritas para indicar vectores.

La ecuación que utilizaremos es pues la de la ley fundamental de la dinámica clásica:

$$\sum_i \mathbf{f}_i = m\mathbf{a}$$

donde las  $\mathbf{f}_i$  denotan las fuerzas aplicadas a la partícula,  $m$  su masa, y  $\mathbf{a}$  su aceleración.

A partir de esta ecuación podemos deducir la trayectoria de nuestra partícula integrando sucesivamente la aceleración. A partir de ahora denotaremos por  $\mathbf{f}$  la suma de las fuerzas:

$$\begin{aligned}\mathbf{a} &= \frac{\mathbf{f}}{m} && \text{donde} && \mathbf{a} = \frac{d\mathbf{v}}{dt} && \text{y} \\ \mathbf{v} &= \frac{d\mathbf{x}}{dt} && \text{lo que implica} && \frac{\mathbf{f}}{m} &= \frac{d^2\mathbf{x}}{dt^2}\end{aligned}$$

Nos encontramos pues delante de una ecuación diferencial de segundo orden.

Es importante notar que, debido a los algoritmos numéricos empleados, podremos tratar sistemas *no globalmente predictivos* (es decir, en todo el intervalo  $t$  de simulación), sistemas que puedan presentar discontinuidades esporádicas en las propias fuerzas (posibilidad de colisiones entre partículas, o de aparición de nuevas fuerzas, permitiendo, por ejemplo el movimiento de ropa debido a la abertura súbita de una puerta). Trabajaremos con intervalos de tiempo  $\Delta t$  (discretización del tiempo), que no tienen porqué ser constantes durante toda la simulación. Los podremos hacer variar para controlar la aparición de efectos deseados, o el error introducido por la misma discretización.

No resolvemos el sistema a la manera de los ejercicios clásicos de mecánica consistentes en encontrar la función  $\mathbf{x}(t)$  de la trayectoria de la

partícula para todo  $t$ . Trabajaremos por intervalos  $\Delta t$ , en los que "resolveremos" la ecuación diferencial del sistema en el sentido de encontrar la posición y velocidad de la partícula al final del intervalo en función de los valores de la partícula al principio del intervalo (condiciones iniciales). Es claro que la razón por la cual guardamos, además de la posición, la velocidad es que la ecuación diferencial para cada partícula es de segundo orden. Su solución, por lo tanto, depende de dos condiciones, y para que estas sean "iniciales" (y no "de contorno") no pueden ser otras que la posición y velocidad. Por lo tanto se caracterizará una partícula por una cuadrupla de posición, masa, velocidad y acumulador de fuerzas, representando este último la suma de todas las fuerzas que se aplican a la partícula.

En resumen:

$$\mathbf{x}_{t+\Delta t} = \varphi(\mathbf{x}_t, \mathbf{v}_t, \mathbf{f}_{\text{en el intervalo } \Delta t})$$

Partícula a  $t$                                   Partícula a  $t + h$

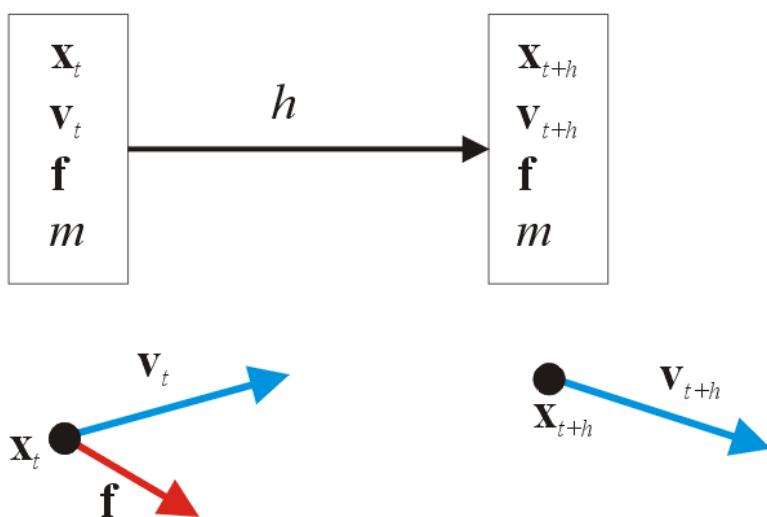


Figura 3.2.1 Cambio de estado de una partícula en el intervalo  $\Delta t = h$ .

En el instante  $t + \Delta t$  calcularemos de nuevo el valor del acumulador de fuerzas para encontrar la futura posición de la partícula. El responsable de resolver la ecuación diferencial en el intervalo  $\Delta t$  será el motor de cálculo, que denominaremos a partir de ahora "solver numérico" o "solver". Existen muchos métodos de integración, unos más rápidos que otros. Generalmente si buscamos velocidad nos encontraremos con poca precisión. En función de la aplicación y del error permitido escogeremos uno u otro.

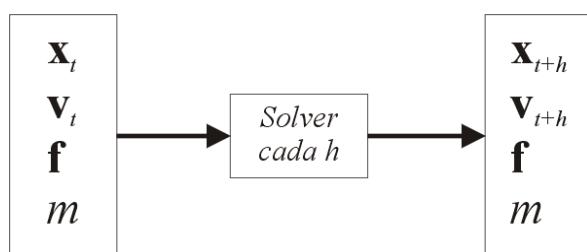


Figura 3.2.2 Acción del solver en cada intervalo  $\Delta t = h$ .

Vamos a considerar ahora sistemas de partículas. En ellos conviene diferenciar las fuerzas internas y las externas.

1. Las fuerzas internas o de interacción son las que se ejercen entre dos o más partículas del sistema. Por ejemplo en sistemas de 2 partículas conectadas por un resorte, la fuerza interna es la fuerza del resorte.
2. Las fuerzas externas son las que se ejercen sobre cada partícula independientemente del estado de las demás. Ejemplos de fuerzas externas son el peso o el viento.

Los pasos a seguir para resolver un sistema de partículas pueden ser los siguientes:

- 1) Calcular las fuerzas internas del sistema sobre cada partícula e inscribir las en su acumulador de fuerzas.
- 2) Calcular igualmente las fuerzas externas y añadirlas al acumulador de fuerzas.
- 3) Calcular la nueva posición de cada partícula con el método de integración, o solver, escogido (aquí es donde se tendrán en cuenta las colisiones).
- 4) Representar gráficamente el sistema de partículas (si es pertinente).
- 5) Reinicializar el valor del acumulador de fuerzas y volver al primer paso.

Esquema:

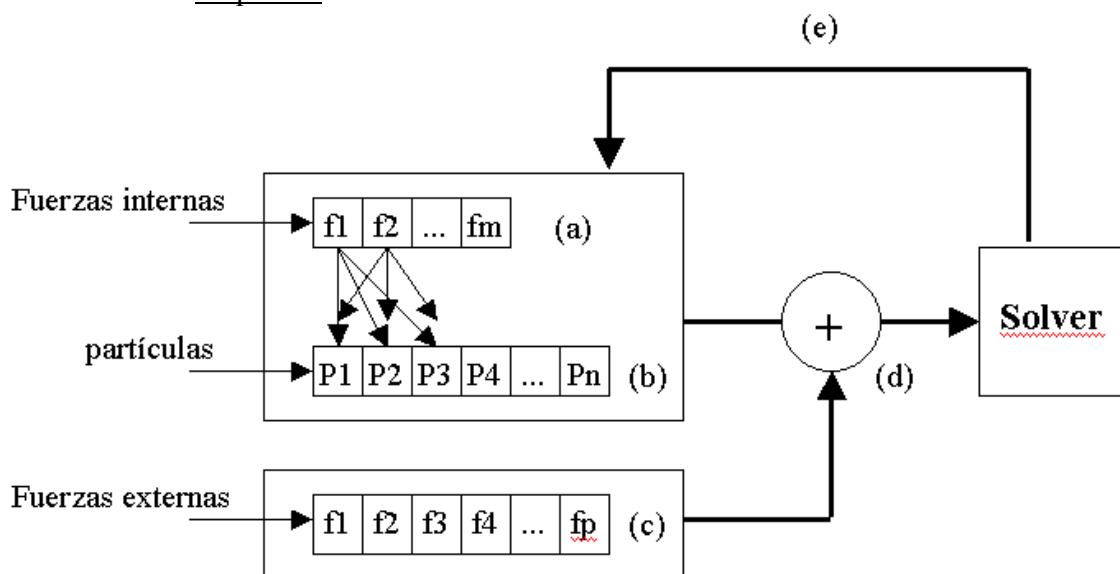


Figura 3.2.3 Diagrama de la estructura de un sistemas de partículas.

- (a) *Tipos de fuerzas internas que aplicamos al sistema (interacción gravitacional, interacción eléctrica...)*
- (b) *Conjunto de partículas de nuestro sistema*
- (c) *Tipos de fuerzas externas que aplicamos al sistema (viento ,peso...).*
- (d) *Sumamos estas fuerzas en el acumulador de cada partícula.*
- (e) *Actualización de las partículas*

Pondremos un ejemplo práctico para visualizar mejor los conceptos anteriores.

Supongamos que nuestro sistema de partículas es un conjunto de  $n$  electrones que giran alrededor de un protón fijo; por ser fijo consideramos que no forma parte del sistema de partículas.

Los electrones se repelen entre sí por tener cargas iguales; cada electrón ejerce una fuerza determinada sobre cada uno de los otros electrones. Esta fuerza es la fuerza interna, y la primera que calculamos (paso 1). Tenemos en total tantas fuerzas como pares de electrones diferentes. Es decir  $C_n^2$  interacciones.

Después tenemos que añadir la fuerza ejercida por el protón sobre cada una de las partículas. Esta es la fuerza externa (en nuestro modelo no depende de la interacción entre partículas). Tendremos  $n$  interacciones (paso 2).

Una vez evaluadas todas las fuerzas podemos calcular la nueva posición de las partículas con ayuda del solver (paso 3). Dependiendo del método de integración, el cálculo será más o menos rápido.

Ahora podemos representar gráficamente las partículas (paso 4) y reinicializar el acumulador de fuerzas (paso 5).

Como podemos ver la resolución de un sistema de partículas es bastante laboriosa y aumenta generalmente (depende de las fuerzas que entran en juego ) muy rápidamente si existe interacción entre las partículas. Es un factor a tener en cuenta a la hora de diseñar sistemas de partículas. Por ello siempre tendremos que intentar minimizar el número de partículas.

## 3.3 Métodos de integración (solver numérico)

### 3.3.1 Introducción a los métodos numéricos de integración

Como se ha visto anteriormente, el solver es el responsable de resolver la ecuación diferencial ordinaria de segundo orden que nos proporciona la nueva posición de la partícula al final de un intervalo  $\Delta t$  que puede ser variable.

La resolución analítica de la ecuación diferencial es muy costosa y rápidamente muy complicada. Como realmente no nos interesa la función en sí sino su valor al final del intervalo utilizamos métodos numéricos de aproximación. Estos métodos son generales a la computación y pueden ser utilizados en muchos otros ámbitos. Existen muchos métodos de integración, es la razón por la cual hemos seleccionado los más utilizados en computación gráfica.

Toda ecuación diferencial ordinaria de orden  $p$  puede escribirse como un sistema de  $p$  ecuaciones diferenciales de primer orden. A partir de ahora trabajaremos siempre con estas últimas. Por ejemplo la ecuación de segundo orden

$$\frac{d^2y}{dx^2} + q(x) \frac{dy}{dx} = r(x)$$

puede escribirse como el sistema de dos ecuaciones de primer orden

$$\frac{dy}{dx} = z(x)$$

$$\frac{dz}{dx} = r(x) - q(x)z(x),$$

donde simplemente añadimos la variable intermedia  $z$ . En nuestro caso concreto de las ecuaciones de segundo orden de la dinámica, la variable intermedia es la velocidad.

De

$$\frac{d^2\mathbf{x}}{dt^2} = \frac{\mathbf{f}}{m}$$

pasamos a

$$\frac{d\mathbf{x}}{dt} = \mathbf{v} \quad \text{y} \quad \frac{d\mathbf{v}}{dt} = \frac{\mathbf{f}}{m}$$

A continuación recordaremos el método de resolución de estas ecuaciones diferenciales por series de Taylor. No es un método estrictamente numérico pero se utiliza como base para los otros métodos. Después veremos el de Euler, el más sencillo pero también el que comete mayor error, y luego los métodos de Runge-Kutta, los más utilizados en simulación gráfica.

Para acabar con los métodos numéricos de integración enumeraré a título informativo otros métodos que no utilizan la filosofía de los anteriores.

### 3.3.2 Serie de Taylor

Como se ha indicado en la introducción, no es un método estrictamente numérico. Resolveremos un ejemplo por serie de Taylor con el fin de entender su utilidad en el diseño de otros métodos.

Consideremos el problema siguiente (en lo que sigue utilizaremos siempre el mismo ejemplo):

$$\frac{dy}{dt} = -2t - y, \quad y(0) = -1$$

La solución analítica,

$$y(t) = -3e^{-t} - 2t + 2$$

se obtiene inmediatamente utilizando métodos estándares de resolución de ecuaciones diferenciales. Desarrollando la solución en serie de Taylor para un valor  $t_0 + h$  donde  $h = \Delta t$ :

$$y(t_0 + h) = y(t_0) + y'(t_0)h + \frac{y''(t_0)}{2!}h^2 + \frac{y'''(t_0)}{3!}h^3 + \dots$$

Podremos ver por su estructura la similitud con los métodos iterativos para la informática y por consecuencia su interés en el tema.

A partir de la ecuación diferencial, y hasta el cuarto orden, se tiene por derivación:

$$\begin{aligned} y' &= -2t - y \\ y'' &= -2 - y' \\ y''' &= -y'' \\ y^{iv} &= -y''' \end{aligned}$$

Luego para  $t_0 = 0$  resulta

$$\begin{aligned} y'(0) &= -y(0) = 1 \\ y''(0) &= -2 - 1 = -3 \\ y'''(0) &= -(-3) = 3 \\ y^{iv}(0) &= -3 \end{aligned}$$

La solución es entonces:

$$y(0 + h) = -1 + 1.0h - 1.5h^2 + 0.5h^3 - 0.125h^4 + \text{error}$$

donde el error se puede escribir como

$$\text{error} = \frac{y''(\xi)}{(5!)} h^5 = O(h^5),$$

en el intervalo  $0 < \xi < h$ , para un determinado valor de  $\xi$ .

Podemos ver que en función del número de términos nuestro error será más o menos grande.

Si utilizamos este método iterativamente, es decir que para calcular el valor de  $t + 2h$  utilizamos el valor de  $t + h$  como condición inicial, obtenemos la estructura básica de resolución de los métodos Euler y Runge-Kutta.

Estos métodos se diferencian básicamente en el número de términos que utilizan de la serie de Taylor.

### 3.3.3 Euler

En la sección anterior hemos visto una manera de resolver ecuaciones diferenciales ordinarias por la serie de Taylor. El problema principal es el cálculo de las derivadas sucesivas de  $y(t)$ . Existen métodos y programás que calculan analíticamente derivadas, pero muchas veces la complicación de estos cálculos crece muy rápidamente. Además para nuestras simulaciones nos interesan métodos rápidos.

La primera aproximación de nuestra función sería la de quedarnos con solo 2 términos de la serie de Taylor. Eliminamos así los problemas de derivación ya que el segundo término solo utiliza  $y'(t)$  y este esta dado por la ecuación. El error cometido disminuye si  $\Delta t = h$  disminuye. Para un valor suficientemente pequeño de  $h$  esta aproximación nos puede ser valida.

$$y(t_0 + h) = y(t_0) + y'(t_0)h,$$

Gráficamente podemos ver que aproximamos el valor  $y(t_0 + h)$  en el punto  $t_0 + h$ , donde  $h = t_1 - t_0$ , por el valor  $y_1$  de la recta tangente a  $y(t)$  en el punto  $t_0$ .

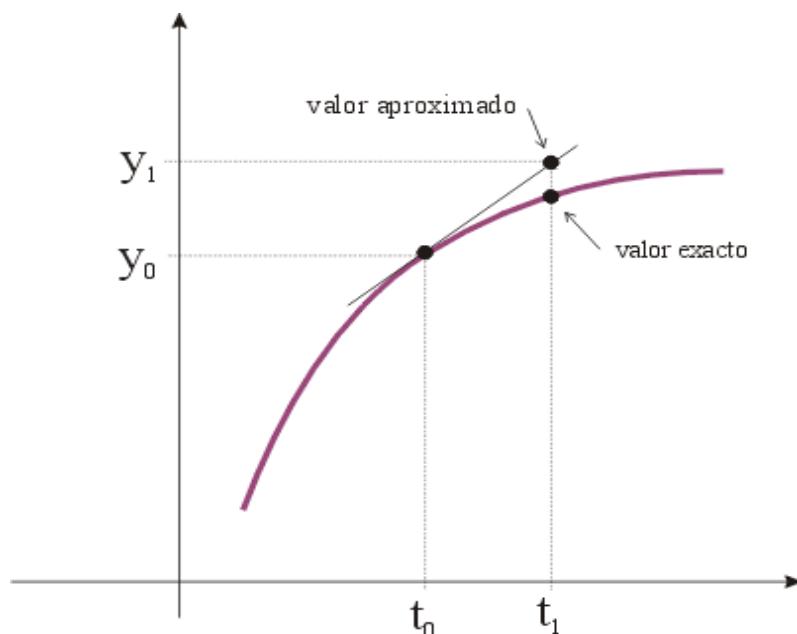


Figura 3.3.2 Ilustración del método de Euler.

Si nuestra función tiene una variación muy rápida necesitamos un  $h$  muy pequeño lo que puede resultar muy lento en muchos casos. Principalmente el método de Euler solo se utiliza en sistemas de fuerzas constantes o en aquellos cuyas soluciones dependen lentamente del tiempo.

Escribamos nuestras ecuaciones de manera que se aproxime más a la posible implementación con un lenguaje de programación:

$$\frac{dy}{dt} = f(t, y),$$

$$y(t_0 + \Delta t) = y(t_0) + f(t_0, y_0)h,$$

o en su forma iterativa:

$$y_{n+1} = y_n + f(t_n, y_n)h$$

Nuestro ejemplo:

$$\frac{dy}{dt} = f(t, y) = 2t - y, \quad y(t_0 = 0) = -1$$

reemplazando la función  $f$  en la ecuación anterior obtenemos:

$$y_{n+1} = y_n + (-2t_n - y_n)h,$$

y para  $t_n = 0$  con  $y_n = -1$ ,

$$y_{n+1} = -1 + 1.0h$$

Nuestro caso de dinámica:

Primero se calcula el valor  $\mathbf{f}$  del acumulador de fuerzas sobre la partícula de masa  $m$  en función de la configuración de nuestro sistema (ver capítulo anterior). Luego se obtiene para cada partícula, su nueva posición y velocidad en el incremento de tiempo  $h$  mediante las relaciones:

$$\begin{aligned}\mathbf{x}_{n+1} &= \mathbf{x}_n + \mathbf{v}_n h \\ \mathbf{v}_{n+1} &= \mathbf{v}_n + \frac{\mathbf{f}}{m} h\end{aligned}$$

Para simulaciones minimamente serias no se suele utilizar este método ya que, como hemos dicho, para mantener un error pequeño hemos de utilizar un incremento de tiempo también muy pequeño.

### 3.3.4 Runge-Kutta 2

Los métodos Runge-Kutta de integración han sido creados por los dos matemáticos alemanes Carle D. T. Runge y Martín W. Kutta. Desarrollaron unos algoritmos para resolver una ecuación diferencial eficientemente, equivalentes a desarrollos en n primeros términos de la serie de Taylor. En este apartado veremos el segundo orden de este método y el cuarto orden en el siguiente; existen órdenes más elevados, que no consideraremos aquí.

El Runge-Kutta 2 equivale a aproximar la solución por los 3 primeros términos de la serie de Taylor.

La idea es obtener el valor  $y_{n+1}$  de la función  $y(t)$  al final del intervalo  $\Delta t = h$  con una media ponderada de 2 estimaciones en la ecuación diferencial  $dy/dt = f(t, y)$ , de la forma:

$$\boxed{\begin{aligned} y_{n+1} &= y_n + ak_1 + bk_2, \\ k_1 &= hf(t_n, y_n), \\ k_2 &= hf(t_n + \alpha h, y_n + \beta k_1) \end{aligned}} \quad (1)$$

A continuación veremos cuáles han de ser los valores de los parámetros  $a, b, \alpha, \beta$  para que obtengamos el mismo resultado que en un desarrollo de Taylor de 3 términos es decir:

$$y_{n+1} = y_n + hf(t_n, y_n) + \frac{f'(t_n, y_n)}{2} h^2 .$$

como  $f(t, y)$  depende de  $t$  a través de sus dos variables, se tiene

$$\frac{df}{dt} = f_t + f_y \frac{dy}{dt} = f_t + f_y f ,$$

y substituyendo ésto en la ecuación anterior resulta:

$$y_{n+1} = y_n + hf_n + h^2 \left( \frac{1}{2} f_t + \frac{1}{2} f_y f \right)_n \quad (2)$$

donde, para toda función  $g(x, y)$  se ha puesto

$$(g)_n = g_n = g(x_n, y_n).$$

Con  $k_1$  y  $k_2$  obtenemos de la aproximación (1)

$$y_{n+1} = y_n + ahf_n + bhf(x_n + \alpha h, y_n + \beta hf_n) \quad (3)$$

Para identificar esta ecuación con (2), desarrollamos el ultimo termino  $f(t_n + \alpha h, y_n + \beta h f_n)$  en su serie de Taylor. Solo utilizaremos su primera derivada ya que solo nos interesan los términos hasta el orden 2 en  $h$ . Como  $f(t, y)$  es una función de 2 variables, su desarrollo en serie de Taylor al primer orden es de la forma:

$$f(t_0 + \Delta x, y_0 + \Delta y) = f(t_0, y_0) + f_t(t_0, y_0)\Delta t + f_y(t_0, y_0)\Delta y ,$$

en consecuencia

$$f(t_n + \alpha h, y_n + \beta h f_n) = f_n + (f_t)_n \alpha h + (f_y)_n \beta h f_n .$$

Substituyendo esta expresión en (3):

$$y_{n+1} = y_n + ahf_n + bh(f_n + (f_t)_n \alpha h + (f_y)_n \beta h f_n))$$

y organizando términos, se tiene:

$$y_{n+1} = y_n + (a + b)hf_n + h^2(\alpha bf_t + \beta bf_y f)_n \quad (4)$$

Las ecuaciones (2) y (4) son equivalentes si y solo si

$$a + b = 1, \quad \alpha = \beta = \frac{1}{2b}, \quad b \neq 0$$

Como solo tenemos tres condiciones para 4 incógnitas, existe una familia a un parámetro de métodos Runge-Kutta 2. Algunos de ellos reciben nombres especiales; así los valores

$$a = b = \frac{1}{2}, \quad \alpha = \beta = 1$$

corresponden al método llamado de Euler modificado, y los valores

$$a = 0, \quad b = 1, \quad \alpha = \beta = \frac{1}{2}$$

al del método llamado de Heun.

Nuestro ejemplo:

$$\frac{dy}{dt} = f(t, y) = -2t - y, \quad y(t_0 = 0) = -1.$$

Resolviendo el problema de forma iterativa, por el método de Euler modificado:

$$y_{n+1} = y_n + \frac{k_1 + k_2}{2}$$

$$k_1 = f(t_n, y_n)h = (-2t_n - y_n)h$$

$$k_2 = f(t_n + h, y_n + k_1)h = (-2(t_n + h) - (y_n + k_1))h;$$

substituyendo y agrupando términos obtenemos

$$y_{n+1} = y_n + -(t_n + y_n)h + \frac{(-2 + 2t_n + y_n)}{2}h^2.$$

Para  $t_0 = 0$  y  $y(0) = -1$  obtenemos

$$y_{n+1} = -1 + 1.0h - 1.5h$$

Vemos que el resultado coincide con el obtenido por Taylor al orden 2.

Nuestro caso de dinámica:

Mostraremos un posible algoritmo para una partícula. La extensión al sistema completo de partículas es directa.

La fuerza que se aplica a una partícula depende de sus condiciones iniciales (posición y velocidad) y del tiempo  $t$ . Denominaremos  $\mathbf{f}(\mathbf{x}_n, \mathbf{v}_n, t_n)$  esa fuerza.

Tenemos como ecuaciones diferenciales:

$$\frac{d\mathbf{x}}{dt} = \mathbf{v} \quad \text{y} \quad \frac{d\mathbf{v}}{dt} = \frac{\mathbf{f}(\mathbf{x}, \mathbf{v}, t)}{m}$$

La extensión del método anterior a dos ecuaciones de primer orden es de la forma:

$$\begin{aligned}\mathbf{x}_{n+1} &= \mathbf{x}_n + a\mathbf{k}_1 + b\mathbf{k}_2 \\ \mathbf{v}_{n+1} &= \mathbf{v}_n + a\mathbf{l}_1 + b\mathbf{l}_2\end{aligned}$$

donde

$$\begin{aligned}\mathbf{k}_1 &= h\mathbf{v}(\mathbf{x}_n, \mathbf{v}_n, t_n), \\ \mathbf{l}_1 &= \frac{h}{m}\mathbf{f}(\mathbf{x}_n, \mathbf{v}_n, t_n), \\ \mathbf{k}_2 &= h\mathbf{v}(\mathbf{x}_n + \beta\mathbf{k}_1, \mathbf{v}_n + \beta\mathbf{l}_1, t_n + \alpha h), \\ \mathbf{l}_2 &= \frac{h}{m}\mathbf{f}(\mathbf{x}_n + \beta\mathbf{k}_1, \mathbf{v}_n + \beta\mathbf{l}_1, t_n + \alpha h).\end{aligned}$$

Como la función  $\mathbf{v}(\mathbf{x}, \mathbf{v}, t)$  no depende ni de  $\mathbf{x}$  ni de  $t$ , y es idénticamente  $\mathbf{v} = \mathbf{v}(\mathbf{v}) = \mathbf{v}$ , las constantes  $\mathbf{k}_1$  y  $\mathbf{k}_2$  resultan ser

$$\mathbf{k}_1 = h\mathbf{v}_n,$$

$$\mathbf{k}_2 = h(\mathbf{v}_n + \beta \mathbf{l}_1) ,$$

A continuación, escribamos la solución de forma más similar a una posible implementación en un lenguaje de programación.

Primero calculamos  $\mathbf{f}_1 = \mathbf{f}(\mathbf{x}_n, \mathbf{y}_n, t_n)$  y se tiene:

$$\mathbf{k}_1 = \mathbf{v}_n h ,$$

$$\mathbf{l}_1 = \frac{\mathbf{f}_1}{m} h ;$$

en primera a aproximación tenemos resulta:

$$\mathbf{x}'_{n+1} = \mathbf{x}_n + \beta \mathbf{k}_1 ,$$

$$\mathbf{v}'_{n+1} = \mathbf{v}_n + \beta \mathbf{l}_1 .$$

A continuación calculamos  $\mathbf{f}_2 = \mathbf{f}(\mathbf{x}'_{n+1}, \mathbf{v}'_{n+1}, t_n + \alpha h)$  y se tiene:

$$\mathbf{k}_2 = \mathbf{v}'_{n+1} h ,$$

$$\mathbf{l}_2 = \frac{\mathbf{f}_2}{m} h .$$

El resultado final es:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + a \mathbf{k}_1 + b \mathbf{k}_2 ,$$

$$\mathbf{v}_{n+1} = \mathbf{v}_n + a \mathbf{l}_1 + b \mathbf{l}_2 .$$

### 3.3.5 Runge-kutta 4

El método de Runge-Kutta 4 es el más utilizado en simulaciones gráficas. Es mucho más preciso que el anterior ya que tiene en cuenta los 5 primeros términos de la serie de Taylor. Lo consigue mediante la media ponderada de 4 incrementos intermedios. Tenemos, como en el de Runge-Kutta 2, toda una familia de soluciones (ahora bi-paramétrica) debido a la diferencia entre el número de parámetros (=13) y el de condiciones (=11). Aquí nos limitaremos a presentar sin demostración el método de Runge-Kutta 4 para los valores más típicos de los parámetros:

$$\boxed{\begin{aligned}y_{n+1} &= y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4), \\k_1 &= hf(t_n, y_n), \\k_2 &= hf(t_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1), \\k_3 &= hf(t_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2), \\k_4 &= hf(t_n + h, y_n + k_3)\end{aligned}}$$

Nuestro ejemplo:

$$\frac{dy}{dt} = f(t, y) = -2t - y, \quad y(t_0 = 0) = -1$$

notación y condiciones iniciales:

$$h = \Delta t, \quad t_0 = 0, \quad y_0 = y(t_0) = -1$$

$$\begin{aligned}k_1 &= (-2t_0 - y_0)h = h \\k_2 &= (-2(t_0 + \frac{1}{2}h) - (y_0 + \frac{1}{2}k_1))h\end{aligned}$$

Substituyendo  $k_1$  y organizando términos

$$\begin{aligned}k_2 &= h - 1.5h^2 \\k_3 &= (-2(t_0 + \frac{1}{2}h) - (y_0 + \frac{1}{2}k_2))h\end{aligned}$$

Substituyendo  $k_2$  y organizando términos

$$k_3 = h - 1.5h^2 + 0.75h^3$$

$$k_4 = (-2(t_0 + h) - (y_0 + k_3))h$$

Substituyendo  $k_3$  y organizando términos

$$k_4 = h - 3h^2 + 1.5h^3 - 0.75h^4$$

La solución es:

$$\begin{aligned} y_{n+1} &= y_0 + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \\ y_{n+1} &= -1 + 1.0h - 1.5h^2 + 0.5h^3 - 0.125h^4 \end{aligned}$$

Vemos que obtenemos la misma solución que la obtenida anteriormente en el apartado [3.3.2] por el desarrollo en serie de Taylor al cuarto orden.

Nuestro caso de dinámica:

Describiremos el algoritmo para una partícula de posición  $\mathbf{x}_n$ , de velocidad  $\mathbf{v}_n$ , de masa  $m$ , en el tiempo  $t_n$ .

Las ecuaciones diferenciales son:

$$\frac{d\mathbf{x}}{dt} = \mathbf{v} \quad \text{y} \quad \frac{d\mathbf{v}}{dt} = \frac{\mathbf{f}(\mathbf{x}, \mathbf{v}, t)}{m}$$

La solución iterativa es:

$$\begin{aligned} \mathbf{x}_{n+1} &= \mathbf{x}_n + \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + \mathbf{k}_3 + \mathbf{k}_4) \\ \mathbf{v}_{n+1} &= \mathbf{v}_n + \frac{1}{6}(\mathbf{l}_1 + 2\mathbf{l}_2 + 2\mathbf{l}_3 + \mathbf{l}_4) \end{aligned}$$

donde

$$\begin{aligned} \mathbf{k}_1 &= h\mathbf{v}_n \\ \mathbf{l}_1 &= h\mathbf{f}(\mathbf{x}_n, \mathbf{y}_n, t_n) \\ \mathbf{k}_2 &= h(\mathbf{v}_n + \frac{1}{2}\mathbf{l}_1) \\ \mathbf{l}_2 &= h\mathbf{f}(\mathbf{x}_n + \frac{1}{2}\mathbf{k}_1, \mathbf{v}_n + \frac{1}{2}\mathbf{l}_1, t + \frac{1}{2}h) \\ \mathbf{k}_3 &= h(\mathbf{v}_n + \frac{1}{2}\mathbf{l}_2) \end{aligned}$$

$$\mathbf{l}_3 = h\mathbf{f}(\mathbf{x}_n + \frac{1}{2}\mathbf{k}_2, \mathbf{v}_n + \frac{1}{2}\mathbf{l}_2, t_n + \frac{1}{2}h)$$

$$\mathbf{k}_4 = h(\mathbf{v}_n + \mathbf{l}_3)$$

$$\mathbf{l}_4 = h\mathbf{f}(\mathbf{x}_n + \mathbf{k}_3, \mathbf{v}_n + \mathbf{l}_3, t_n + h)$$

A continuación escribiremos la solución de forma más similar a la posible implementación en un lenguaje de programación.

Primero calculamos  $\mathbf{f}_1 = \mathbf{f}(\mathbf{x}_n, \mathbf{v}_n, t_n)$  y se tiene

$$\mathbf{k}_1 = \mathbf{v}_n h \quad ,$$

$$\mathbf{l}_1 = \frac{\mathbf{f}_1}{m} h \quad ;$$

en primera aproximación resulta:

$$\mathbf{x}'_{n+1} = \mathbf{x}_n + \frac{1}{2}\mathbf{k}_1 \quad ,$$

$$\mathbf{v}'_{n+1} = \mathbf{v}_n + \frac{1}{2}\mathbf{l}_1 \quad .$$

A continuación calculamos  $\mathbf{f}_2 = \mathbf{f}(\mathbf{x}'_{n+1}, \mathbf{v}'_{n+1}, t_n + \frac{1}{2}h)$  y se tiene

$$\mathbf{k}_2 = \mathbf{v}'_{n+1} h \quad ,$$

$$\mathbf{l}_2 = \frac{\mathbf{f}_2}{m} h \quad ;$$

en segunda aproximación resulta:

$$\mathbf{x}''_{n+1} = \mathbf{x}_n + \frac{1}{2}\mathbf{k}_2 \quad ,$$

$$\mathbf{v}''_{n+1} = \mathbf{v}_n + \frac{1}{2}\mathbf{l}_2 \quad .$$

Calculamos ahora  $\mathbf{f}_3 = \mathbf{f}(\mathbf{x}''_{n+1}, \mathbf{v}''_{n+1}, t_n + \frac{1}{2}h)$ , y se tiene

$$\mathbf{k}_3 = \mathbf{v}''_{n+1} h \quad .$$

$$\mathbf{l}_3 = \frac{\mathbf{f}_3}{m} h \quad ;$$

En tercera aproximación resulta:

$$\begin{aligned}\mathbf{x}_{n+1}''' &= \mathbf{x}_n + \mathbf{k}_3 & , \\ \mathbf{v}_{n+1}''' &= \mathbf{v}_n + \mathbf{l}_3 & .\end{aligned}$$

Calculamos finalmente  $\mathbf{f}_4 = \mathbf{f}(\mathbf{x}_{n+1}''', \mathbf{v}_{n+1}''', t_n + h)$  y se tiene:

$$\begin{aligned}\mathbf{k}_4 &= \mathbf{v}_{n+1}''' h & , \\ \mathbf{l}_4 &= \frac{\mathbf{f}_4}{m} h & .\end{aligned}$$

La solución final de la nueva posición y velocidad es:

$$\begin{aligned}\mathbf{x}_{n+1} &= \mathbf{x}_n + \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4) \\ \mathbf{v}_{n+1} &= \mathbf{v}_n + \frac{1}{6}(\mathbf{l}_1 + 2\mathbf{l}_2 + 2\mathbf{l}_3 + \mathbf{l}_4)\end{aligned}$$

### 3.3.6 Otros métodos

Los métodos Runge-Kutta (y de Euler como caso particular) se llaman métodos de un solo paso porque solo utilizan la información calculada en la iteración anterior. Son útiles cuando empezamos el algoritmo ya que solo disponemos de las condiciones iniciales. Una vez calculado un cierto número de iteraciones disponemos de más información sobre la función y sus derivadas en los puntos anteriores. Si somos capaces de guardar esta información, la podremos utilizar para afinar las iteraciones futuras: es el caso de los métodos multi-pasos.

Nos quedaremos aquí simplemente con esta idea sobre dichos métodos y enumeraré dos de los algoritmos multi-pasos más conocidos:

- Método de Milne (en ciertas ocasiones es inestable)
- Método de Adams-Moulton (más estable que el anterior)

Existen también otros métodos llamados implícitos para la resolución de ecuaciones diferenciales; los Runge-Kutta se llaman métodos explícitos. La diferencia entre ambos reside en la manera de escribir la ecuación, de forma explícita o implícita:

$$\begin{aligned}y' &= f(x, y) \text{ , forma explícita.} \\f(x, y, y') &= 0 \text{ , forma implícita.}\end{aligned}$$

y de realizar consecuentemente los cálculos.

## 3.4 Fuerzas

Recordaremos aquí algunas de las fuerzas más utilizadas en nuestros sistemas de partículas.

### 3.4.1 Viscosidad

Esta fuerza de sentido contrario al del movimiento, y por lo tanto frenándolo, solo depende de la velocidad de la partícula, y tiene por expresión:

$$\mathbf{f} = -k\mathbf{v}$$

En simulaciones de ropa (como veremos en los ejemplo de sistemas de partículas) se puede utilizar para representar la fuerza de rozamiento que ejerce el aire.

### 3.4.2 Gravitacional

Esta fuerza de interacción entre dos partículas, de masa  $m_1$ ,  $m_2$  y situadas en  $\mathbf{x}_1$ ,  $\mathbf{x}_2$  respectivamente, tiene por expresión, sobre la partícula de masa  $m_2$ :

$$\mathbf{f} = -G \frac{m_1 m_2}{|\mathbf{x}_2 - \mathbf{x}_1|^2} \times \frac{(\mathbf{x}_2 - \mathbf{x}_1)}{|\mathbf{x}_2 - \mathbf{x}_1|}$$

donde  $G = 6.67259 \times 10^{-11} \text{ Nm}^2 \text{ kg}^{-2}$ .

El primer término de la fórmula, salvo signo, es el módulo de la fuerza, mientras que el segundo indica su dirección (vector unitario formado por las dos posiciones de las partículas).

Obviamente la partícula de masa  $m_1$  se ve sometida a la fuerza  $(-\mathbf{f})$  por la masa  $m_2$ .

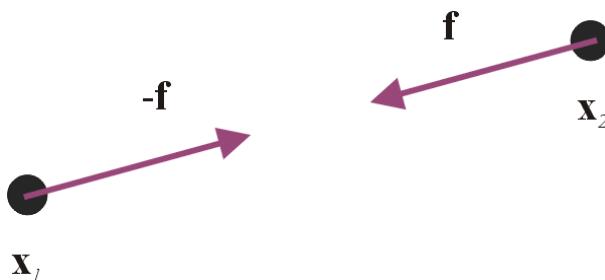


Figura 3.4.2 Representación de las fuerzas de gravedad

### 3.4.3 Eléctrica

Esta fuerza, de interacción entre dos partículas, es de la misma forma que la de gravedad pero depende de la carga de cada partícula y no de su masa. Puede ser repulsiva o atractiva en función del signo de las cargas, y tiene por expresión, sobre la partícula de carga  $q_2$ :

$$\mathbf{f} = C \frac{q_2 q_1}{|\mathbf{x}_2 - \mathbf{x}_1|^2} \times \frac{(\mathbf{x}_2 - \mathbf{x}_1)}{|\mathbf{x}_2 - \mathbf{x}_1|}$$

donde  $C = 8,9875 \cdot 10^9 \text{ Nm}^2\text{C}^{-2}$ .

El primer término, salvo signo es el módulo de la fuerza y el  $\times$  el segundo indica su dirección.

Como para el caso gravitacional, la partícula de carga  $q_1$  se ve sometida a  $(-\mathbf{f})$  por la carga  $q_2$ .

Se tiene el esquema siguiente:

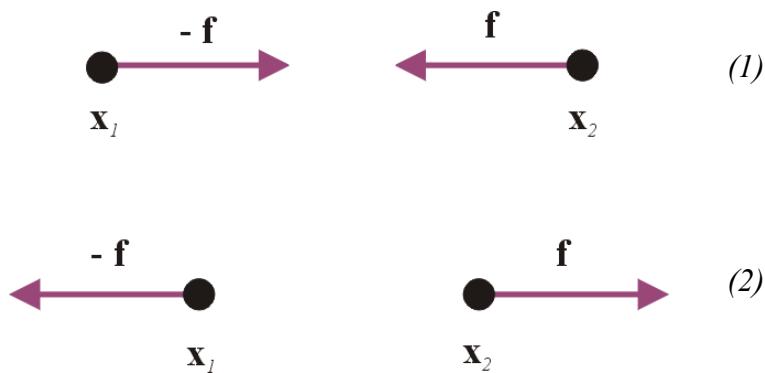


Figura 3.4.3  
 (1) Dos partículas con cargas de signo contrario ( $q_1 q_2 < 0$ )  
 (2) Dos partículas con cargas de mismo signo ( $q_1 q_2 > 0$ )

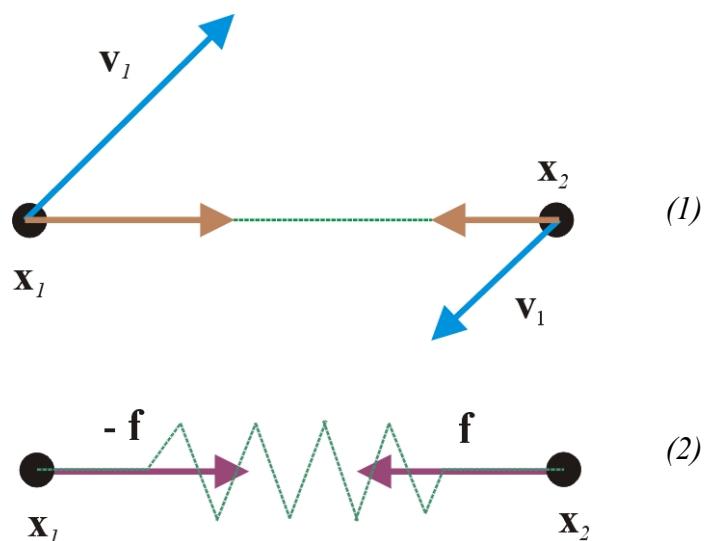
### 3.4.4 Elástica amortiguada

El resorte es una de las fuerzas más utilizadas para modelar objetos elásticos como ropa o cuerdas. Un resorte está caracterizado por una constante  $k_s$  de elasticidad, una longitud  $r$  en reposo y una constante  $k_d$  de amortiguamiento. Entre dos partículas de posiciones  $\mathbf{x}_1$ ,  $\mathbf{x}_2$  y velocidades respectivas  $\mathbf{v}_1$ ,  $\mathbf{v}_2$ , un resorte ejerce sobre la partícula en  $x_2$  una fuerza de expresión:

$$\mathbf{f} = - \left[ k_s (|\mathbf{x}_2 - \mathbf{x}_1| - r) + k_d \frac{(\mathbf{v}_2 - \mathbf{v}_1)(\mathbf{x}_2 - \mathbf{x}_1)}{|\mathbf{x}_2 - \mathbf{x}_1|} \right] \frac{(\mathbf{x}_2 - \mathbf{x}_1)}{|\mathbf{x}_2 - \mathbf{x}_1|}$$

El primer término de la fórmula, salvo signo, es el módulo de la fuerza, y el segundo indica su dirección. En la expresión del primer término se distingue la fuerza puramente elástica, dada por el primer sumando, y que depende exclusivamente de la posición relativa de las dos partículas. El segundo sumando representa la fuerza de amortiguamiento, dependiente de la proyección de la velocidad relativa sobre la dirección de interacción.

Como en los casos anteriores la partícula en  $\mathbf{x}_1$  se ve sometida a la fuerza  $(-\mathbf{f})$ .



*Figura 3.4.4  
 (1) Proyección de las velocidades sobre  $(\mathbf{x}_2 - \mathbf{x}_1)$   
 (2) Aplicación de las fuerzas*

### 3.5 Funciones de energía

Algunas veces necesitamos que nuestro sistema tienda a una cierta configuración. La fuerzas de partida pueden no ser suficientes para llevar el sistema a es configuración y tendremos que diseñar otras fuerzas para ese fin. Si somos capaces de asociar al de partículas una función  $\mathbf{C}(\mathbf{x}_1, \mathbf{x}_2, \dots)$  (donde  $\mathbf{x}_i$  es la posición de la partícula  $i$ ), a valores no necesariamente escalares, que tienda a cero para la configuración deseada, podremos encontrar esas fuerzas. A una tal función se le llama *función de comportamiento*.

Por ejemplo en el caso de un resorte donde queremos que una partícula de posición  $\mathbf{x}_1$  se quede a la distancia  $r$  de otra partícula  $\mathbf{x}_2$ , la función de comportamiento podría ser  $C(\mathbf{x}_1, \mathbf{x}_2) = |\mathbf{x}_1 - \mathbf{x}_2| - r$ . Denotaremos la función de comportamiento por  $C$  si es una función escalar y por  $\mathbf{C}$  en cualquier otro caso, en particular si es una función vectorial.

Para obtener a partir de la función de comportamiento  $\mathbf{C}(\mathbf{x}_1, \mathbf{x}_2, \dots)$  una fuerza se sigue un procedimiento muy simple. Se define primero una función escalar de energía potencial

$$E = \frac{k_s}{2} \mathbf{C} \cdot \mathbf{C},$$

donde  $k_s$  es una constante de control y  $\mathbf{C} \cdot \mathbf{C}$  designa el cuadrado escalar de  $\mathbf{C}$  (producto ordinario si  $C$  es una función escalar, producto escalar si  $\mathbf{C}$  es una función vectorial). La fuerza resultante sobre la partícula  $p_i$  de coordenadas  $\mathbf{x}_i$  viene dada por el gradiente de la energía, es decir la derivada parcial respecto de  $\mathbf{x}_i$  de la función de energía  $E$

$$\mathbf{f}_i = -\frac{\partial E}{\partial \mathbf{x}_i} = -k_s \mathbf{C} \frac{\partial \mathbf{C}}{\partial \mathbf{x}_i} \quad (1)$$

Frecuentemente  $\mathbf{C}$  es un vector (función vectorial) y entonces  $\mathbf{C} \cdot \partial \mathbf{C} / \partial \mathbf{x}_i$  designa el producto de  $\mathbf{C}$  por la transpuesta de la matriz jacobiana  $\partial \mathbf{C} / \partial \mathbf{x}_i$ . Podemos pensar esta fuerza como la de un resorte que obliga al sistema a tender a  $\mathbf{C} = 0$ . Cuando una función de comportamiento depende de más de una partícula, tendremos una fuerza diferente para cada una de ellas ya que derivaremos  $\mathbf{C}$  respecto de cada una de las variables.

La fuerza así obtenida no es realmente la que queremos. Debido a la ausencia de pérdida de energía (por derivar del potencial  $E$ , el sistema es conservativo), nuestro sistema no tendería realmente hacia  $\mathbf{C} = 0$ , sino que oscilaría alrededor de ese valor. Tenemos que añadirle un término de amortiguamiento dependiente de  $d\mathbf{C}/dt$ . Se suele tomar una expresión modificada de la forma:

$$\mathbf{f}_i = -\left( k_s \mathbf{C} + k_d \frac{d\mathbf{C}}{dt} \right) \frac{\partial \mathbf{C}}{\partial \mathbf{x}_i} \quad (2)$$

donde  $k_d$  es una constante de amortiguamiento.

A continuación mostraremos un ejemplo con un caso muy simple de función  $\mathbf{C}$  vectorial y otro de función  $C$  escalar.

### Ejemplo 1:

Supongamos que queremos que la posición de dos partículas coincida. Nuestra función de comportamiento podría ser  $\mathbf{C} = \mathbf{x}_1 - \mathbf{x}_2$  ya que obviamente tiende a 0 como deseamos.  $\mathbf{C}$  es vectorial por lo tanto tenemos explícitamente:

$$\begin{aligned} C_x &= x_1 - x_2, \\ C_y &= y_1 - y_2, \\ C_z &= z_1 - z_2, \end{aligned}$$

donde  $\mathbf{C} = (C_x, C_y, C_z)$ ,  $\mathbf{x}_1 = (x_1, y_1, z_1)$ ,  $\mathbf{x}_2 = (x_2, y_2, z_2)$ .

Para una función vectorial cualquiera se tiene

$$\frac{\partial \mathbf{C}}{\partial \mathbf{x}_i} = \begin{pmatrix} \frac{\partial C_x}{\partial x_i} & \frac{\partial C_y}{\partial x_i} & \frac{\partial C_z}{\partial x_i} \\ \frac{\partial C_x}{\partial y_i} & \frac{\partial C_y}{\partial y_i} & \frac{\partial C_z}{\partial y_i} \\ \frac{\partial C_x}{\partial z_i} & \frac{\partial C_y}{\partial z_i} & \frac{\partial C_z}{\partial z_i} \end{pmatrix}$$

de modo que para nuestro caso resulta,  $\mathbf{I}$  denotando la matriz identidad:

$$\frac{\partial \mathbf{C}}{\partial \mathbf{x}_1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \mathbf{I}, \quad \frac{\partial \mathbf{C}}{\partial \mathbf{x}_2} = \begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix} = -\mathbf{I}$$

La derivación respecto al tiempo de  $\mathbf{C}$  es

$$\frac{d\mathbf{C}}{dt} = \frac{d\mathbf{x}_1}{dt} - \frac{d\mathbf{x}_2}{dt} = \mathbf{v}_1 - \mathbf{v}_2.$$

donde  $\mathbf{v}_i = d\mathbf{x}_i / dt$  es la velocidad de la partícula  $i$ . Substituyendo estas expresiones en la ecuación (2) de la fuerza global obtenemos

$$\begin{aligned} \mathbf{f}_1 &= -k_s(\mathbf{x}_1 - \mathbf{x}_2) - k_d(\mathbf{v}_1 - \mathbf{v}_2), \\ \mathbf{f}_2 &= k_s(\mathbf{x}_1 - \mathbf{x}_2) + k_d(\mathbf{v}_1 - \mathbf{v}_2), \end{aligned}$$

que es la fuerza de un resorte amortiguado de tamaño cero en reposo.

Ejemplo 2:

En este ejemplo consideramos la función escalar  $C = |\mathbf{x}_1 - \mathbf{x}_2| - r$ . Podría también ser una función de comportamiento de un resorte. Poniendo  $\mathbf{l} = \mathbf{x}_1 - \mathbf{x}_2$  podemos escribir  $C = |\mathbf{l}| - r$ , o explícitamente:

$$C = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2} - r$$

donde  $\mathbf{x}_1 = (x_1, y_1, z_1)$ , y  $\mathbf{x}_2 = (x_2, y_2, z_2)$ .

De forma general se tiene, para una función escalar:

$$\frac{\partial C}{\partial \mathbf{x}_i} = \begin{pmatrix} \frac{\partial C}{\partial x_i} \\ \frac{\partial C}{\partial y_i} \\ \frac{\partial C}{\partial z_i} \end{pmatrix}$$

lo que en nuestro caso da:

$$\frac{\partial C}{\partial \mathbf{x}_1} = \begin{pmatrix} \frac{(x_1 - x_2)}{|\mathbf{l}|} \\ \frac{(y_1 - y_2)}{|\mathbf{l}|} \\ \frac{(z_1 - z_2)}{|\mathbf{l}|} \end{pmatrix} = \frac{\mathbf{x}_1 - \mathbf{x}_2}{|\mathbf{l}|} = \frac{\mathbf{l}}{|\mathbf{l}|}$$

y

$$\frac{\partial C}{\partial \mathbf{x}_2} = \begin{pmatrix} -\frac{(x_1 - x_2)}{|\mathbf{l}|} \\ -\frac{(y_1 - y_2)}{|\mathbf{l}|} \\ -\frac{(z_1 - z_2)}{|\mathbf{l}|} \end{pmatrix} = -\frac{(\mathbf{x}_1 - \mathbf{x}_2)}{|\mathbf{x}|} = -\frac{\mathbf{l}}{|\mathbf{l}|}$$

Tenemos también

$$\frac{dC}{dt} = \frac{\partial C}{\partial \mathbf{l}} \cdot \frac{d\mathbf{l}}{dt}$$

Pero de lo anterior se deduce fácilmente que

$$\frac{\partial C}{\partial \mathbf{l}} = \frac{\mathbf{l}}{|\mathbf{l}|}$$

y por otra parte

$$\frac{d\mathbf{l}}{dt} = \frac{d(\mathbf{x}_1 - \mathbf{x}_2)}{dt} = \mathbf{v}_1 - \mathbf{v}_2$$

donde  $\mathbf{v}_i$  es la velocidad de la partícula  $i$  lo que da:

$$\frac{\partial C}{\partial t} = (\mathbf{v}_1 - \mathbf{v}_2) \cdot \frac{\mathbf{l}}{|\mathbf{l}|} .$$

La expresión final de la fuerza es pués:

$$\begin{aligned}\mathbf{f}_1 &= - \left( k_s C + k_d (\mathbf{v}_1 - \mathbf{v}_2) \cdot \frac{\mathbf{l}}{|\mathbf{l}|} \right) \frac{\mathbf{l}}{|\mathbf{l}|} \\ \mathbf{f}_2 &= \left( k_s C + k_d (\mathbf{v}_1 - \mathbf{v}_2) \cdot \frac{\mathbf{l}}{|\mathbf{l}|} \right) \frac{\mathbf{l}}{|\mathbf{l}|}\end{aligned}$$

## 3.6 Colisiones

Como se dijo en la introducción solo veremos colisiones con planos y triángulos. Trabajaremos en un espacio afín  $(O, \vec{i}, \vec{j}, \vec{k})$ .

### 3.6.1 Colisión con un plano

Separamos el tratamiento de colisiones principalmente en dos etapas. La primera consiste en determinar la existencia o no de colisión y, en caso afirmativo encontrar el punto de colisión sobre el plano. La segunda consiste en aplicar en consecuencia el criterio de colisión elegido: choque elástico, amortiguados, adhesión, penetración más o menos profunda, etc. Aquí solo veremos 2 casos, la colisión puramente elástica y la amortiguada en la dirección normal al plano.

#### 3.6.1.1 Condición de existencia de una colisión

Sea  $M$  un plano de ecuación  $M \equiv \mathbf{n} \cdot \mathbf{x} = k$  (donde  $\mathbf{n}$  es el vector normal unitario al plano) y  $\mathbf{x}_n$  la posición de una partícula en la iteración  $n$ . Supongamos la partícula exterior al plano, es decir  $\mathbf{n} \cdot \mathbf{x}_n \neq k$ .

Diremos que existe *colisión* en la iteración  $n+1$  entre la partícula y el plano si este contiene la posición  $\mathbf{x}_{n+1}$  o la separa de la  $\mathbf{x}_n$ .

Como la condición de que  $M$  contiene  $\mathbf{x}_{n+1}$  es obviamente  $\mathbf{n} \cdot \mathbf{x}_{n+1} = k$ , y la condición de que  $M$  separa  $\mathbf{x}_{n+1}$  de  $\mathbf{x}_n$  es que  $\mathbf{n} \cdot \mathbf{x}_n - k < 0$  y  $\mathbf{n} \cdot \mathbf{x}_{n+1} - k > 0$  sean de signo opuestos (puesto que el plano  $\mathbf{n} \cdot \mathbf{x} - k = 0$  divide el espacio en dos regiones disjuntas  $\mathbf{n} \cdot \mathbf{x} - k > 0$  y  $\mathbf{n} \cdot \mathbf{x} - k < 0$ ), la *condición general de colisión* es :

$$(n \cdot x_n - k)(n \cdot x_{n+1} - k) \leq 0$$

donde la desigualdad estricta tiene lugar si, y solo si,  $\mathbf{x}_{n+1}$  es exterior al plano.

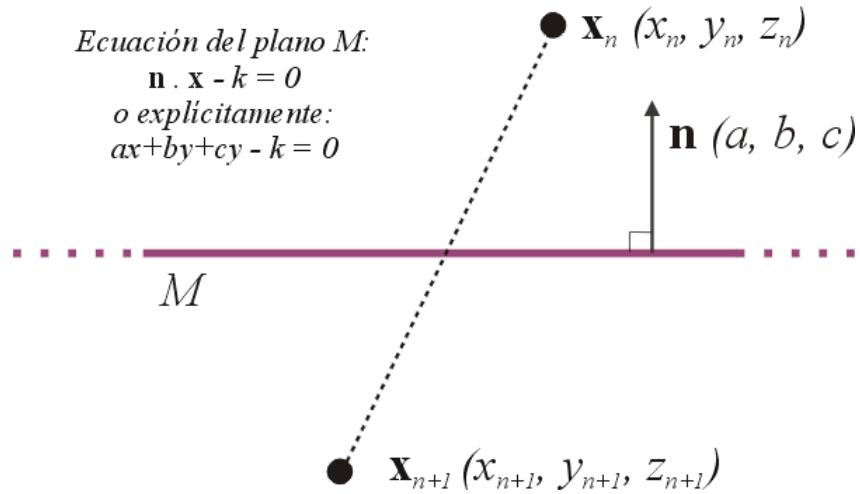


Figura 3.6.1.1

De forma explícita si  $M \equiv ax + by + cz - k = 0$  donde  $\mathbf{n} = (a, b, c)$  y  $a^2 + b^2 + c^2 = 1$ , la condición general de colisión con el plano  $M$  en la iteración de  $\mathbf{x}_n = (x_n, y_n, z_n)$  a  $\mathbf{x}_{n+1} = (x_{n+1}, y_{n+1}, z_{n+1})$  se escribe:

$$(ax_{n+1} + by_{n+1} + cz_{n+1} - k)(ax_n + by_n + cz_n - k) \leq 0$$

### 3.6.1.2 Cómo encontrar el punto de colisión

El punto de colisión es, por definición, el punto de intersección de la recta  $r_{n+1} \equiv \mathbf{x}_{n+1} + t(\mathbf{x}_{n+1} - \mathbf{x}_n)$  y el plano  $M$ .

La ecuación de la recta  $r_{n+1}$  se puede expresar de forma paramétrica como el lugar geométrico de los puntos  $\mathbf{x}(x, y, z)$  que verifican:

$$\mathbf{x} = (\mathbf{x}_{n+1} - \mathbf{x}_n)t + \mathbf{x}_n$$

El valor de  $t$  para el cual  $\mathbf{x}$  está sobre el plano se obtiene imponiendo que  $\mathbf{x}$  verifique su ecuación  $\mathbf{n} \cdot \mathbf{x} = k$ , es decir:

$$\mathbf{n} \cdot [(\mathbf{x}_{n+1} - \mathbf{x}_n)t + \mathbf{x}_n] = k$$

Resulta pues:

$$t = \frac{(k - \mathbf{n} \cdot \mathbf{x}_n)}{\mathbf{n} \cdot (\mathbf{x}_{n+1} - \mathbf{x}_n)}$$

donde es claro que  $\mathbf{n} \cdot (\mathbf{x}_{n+1} - \mathbf{x}_n)$  nunca se anula en caso de colisión, por ser  $\mathbf{x}_n$  exterior al plano.

Substituyendo el valor de  $t$  en la ecuación de la recta  $r_{n+1}$  obtenemos las coordenadas del punto  $\mathbf{x}_c$  de colisión, es decir

$$\boxed{\mathbf{x}_c = \frac{1}{\mathbf{n} \cdot (\mathbf{x}_{n+1} - \mathbf{x}_n)} [(k - \mathbf{n} \cdot \mathbf{x}_n) \mathbf{x}_{n+1} - (k - \mathbf{n} \cdot \mathbf{x}_{n+1}) \mathbf{x}_n]}$$

### 3.6.1.3 Colisión elástica pura

En una colisión elástica pura no hay pérdida de energía. Si denotamos por  $\mathbf{x}'_{n+1}$  y  $\mathbf{v}'_{n+1}$  las nuevas posición y velocidad de la partícula en ausencia de colisión (sin rebote) y  $\mathbf{x}_{n+1}$ ,  $\mathbf{v}_{n+1}$  la posición y velocidad debida al rebote elástico, un resultado conocido indica que  $\mathbf{x}_{n+1}$  y  $\mathbf{v}_{n+1}$  son los simétricos de  $\mathbf{x}'_{n+1}$  y  $\mathbf{v}'_{n+1}$  respecto al plano M. Se tiene pues el diagrama siguiente:

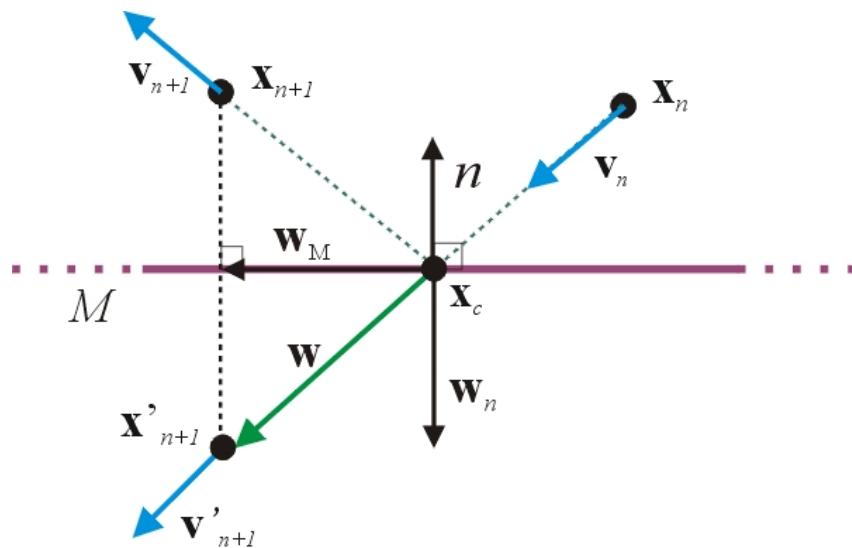


Figura 3.6.1.3

y las relaciones

$$\begin{aligned}\mathbf{x}'_{n+1} - \mathbf{x}_c &= \mathbf{w} = (\mathbf{w} \cdot \mathbf{n})\mathbf{n} + \mathbf{w}_M \\ \mathbf{x}_{n+1} - \mathbf{x}_c &= -(\mathbf{w} \cdot \mathbf{n})\mathbf{n} + \mathbf{w}_M\end{aligned},$$

ya que  $\mathbf{w}_n = (\mathbf{w} \cdot \mathbf{n})\mathbf{n}$ .

Restando se tiene

$$\mathbf{x}_{n+1} - \mathbf{x}'_{n+1} = -2(\mathbf{w} \cdot \mathbf{n})\mathbf{n}$$

donde  $\mathbf{w} \cdot \mathbf{n} = (\mathbf{x}'_{n+1} - \mathbf{x}_c) \cdot \mathbf{n}$ . Siendo  $\mathbf{x}_c$  un punto del plano  $M$ , sabemos que  $\mathbf{x}_c \cdot \mathbf{n} = k$ , luego:

$$\mathbf{w} \cdot \mathbf{n} = (\mathbf{x}'_{n+1} \cdot \mathbf{n}) - k$$

y por lo tanto

$$\boxed{\mathbf{x}_{n+1} = \mathbf{x}'_{n+1} - 2[(\mathbf{x}'_{n+1} \cdot \mathbf{n}) - k]\mathbf{n}}$$

De forma similar, para la velocidad resulta:

$$\begin{aligned}\mathbf{v}'_{n+1} &= (\mathbf{v}'_{n+1} \cdot \mathbf{n})\mathbf{n} + \mathbf{v}_M \\ \mathbf{v}_{n+1} &= -(\mathbf{v}'_{n+1} \cdot \mathbf{n})\mathbf{n} + \mathbf{v}_M\end{aligned}$$

luego

$$\mathbf{v}_{n+1} - \mathbf{v}'_{n+1} = -2(\mathbf{v}'_{n+1} \cdot \mathbf{n})\mathbf{n}$$

y por lo tanto

$$\boxed{\mathbf{v}_{n+1} = \mathbf{v}'_{n+1} - 2(\mathbf{v}'_{n+1} \cdot \mathbf{n})\mathbf{n}}$$

### 3.6.1.4 Colisión amortiguada en la normal al plano

Consideramos ahora una colisión con pérdida de energía pero solamente en la componente normal al plano. Si  $\mathbf{x}''_{n+1}$  es el punto resultado de una colisión puramente elástica (ver caso anterior), el problema consiste en encontrar el punto  $\mathbf{x}_{n+1}$  donde, como podemos ver en la figura siguiente, solo se modifica su coordenada normal.

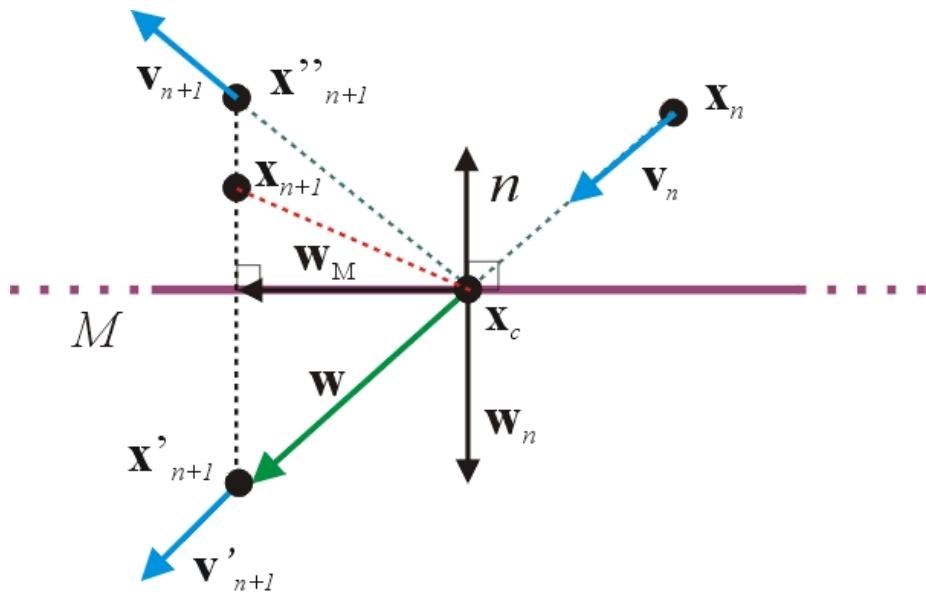


Figura 3.6.1.4

Si denotamos por  $\alpha$  el coeficiente de amortiguamiento correspondiente a las componentes normales de  $x_{n+1}$  y  $v_{n+1}$  es claro entonces que, de acuerdo con las expresiones del caso anterior, se tiene:

$$\boxed{\begin{aligned}\mathbf{x}_{n+1} &= \mathbf{x}'_{n+1} - (1 + \alpha)[(\mathbf{x}'_{n+1} \cdot \mathbf{n}) - k]\mathbf{n} \\ \mathbf{v}_{n+1} &= \mathbf{v}'_{n+1} - (1 - \alpha)[(\mathbf{v}'_{n+1} \cdot \mathbf{n}) - k]\mathbf{n}\end{aligned}}$$

### 3.6.2 Colisión con un triángulo

Un triángulo define un plano y por lo tanto, lo visto anteriormente contiene parcialmente el estudio de la colisión que nos ocupa, es decir la de una partícula con un triángulo.

#### 3.6.2.1 Condición de existencia de la colisión

Una colisión con un triángulo implica una colisión con el plano que él define. Hay colisión con el triángulo si el punto  $x_c$  de intersección con dicho plano anteriormente estudiado, pertenecen al triángulo. El problema es pues el de obtener las condiciones que aseguran dicha situación.

Este problema se resuelve con una idea muy simple. Si la suma de las tres áreas de los triángulos formados por el punto de intersección  $x_c$  y las aristas del triángulo dado es igual que la área de dicho triángulo, nuestro punto de

intersección pertenecerá al triángulo, y por lo tanto habrá colisión. En caso contrario nuestro punto estará fuera, y no habrá colisión.

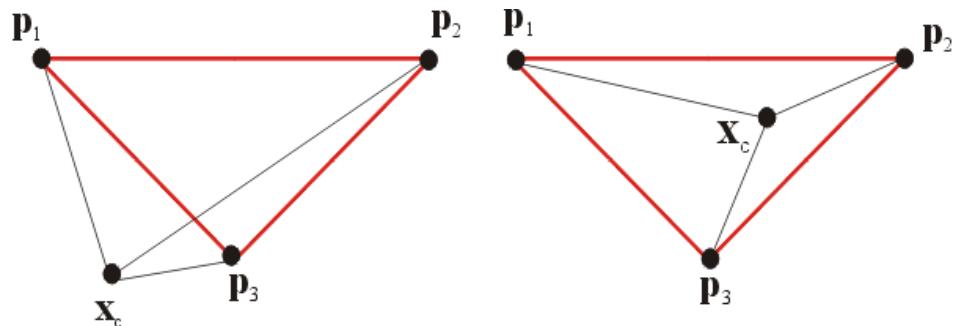


Figura 3.6.2.1

- (1) El punto está fuera del triángulo: la suma de las áreas parciales es más grande que el área del triángulo rojo.
- (2) La suma de las áreas parciales es igual que el área del triángulo rojo.

Denotando por  $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$  los vértices del triángulo dado, por  $\mathbf{x}_c$  el punto de intersección con el plano  $M$ , y por  $A(\mathbf{p}, \mathbf{q}, \mathbf{r})$  el área del triángulo de vértices  $\mathbf{p}, \mathbf{q}, \mathbf{r}$ , se tiene pues la relación:

$$A(\mathbf{x}_c, \mathbf{p}_1, \mathbf{p}_2) + A(\mathbf{x}_c, \mathbf{p}_2, \mathbf{p}_3) + A(\mathbf{x}_c, \mathbf{p}_3, \mathbf{p}_1) - A(\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3) \geq 0$$

donde la igualdad tiene lugar si, y solo si,  $\mathbf{x}_c$  es interior al triángulo  $\{\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3\}$ , es decir, si la partícula en cuestión colisiona con él.

Como el módulo del producto vectorial de dos vectores es el doble del área del triángulo que forman, se tiene

$$A(\mathbf{p}, \mathbf{q}, \mathbf{r}) = \frac{1}{2} |(\mathbf{p} - \mathbf{q}) \times (\mathbf{q} - \mathbf{r})|$$

Recordemos que, para dos vectores  $\mathbf{u} = (u_x, u_y, u_z)$  y  $\mathbf{v} = (v_x, v_y, v_z)$ , es:

$$\mathbf{u} \times \mathbf{v} = \begin{vmatrix} i & j & k \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{vmatrix}$$

y por lo tanto

$$|\mathbf{u} \times \mathbf{v}|^2 = (u_x v_y - u_y v_x)^2 + (u_y v_z - u_z v_y)^2 + (u_z v_x - u_x v_z)^2$$

Así, denotando los lados del triángulo por

$$\mathbf{a}_1 \equiv \mathbf{p}_1 - \mathbf{p}_2, \quad \mathbf{a}_2 \equiv \mathbf{p}_2 - \mathbf{p}_3, \quad \mathbf{a}_3 \equiv \mathbf{p}_3 - \mathbf{p}_1,$$

la condición de colisión con el triángulo puede escribirse

$$|(\mathbf{x}_c - \mathbf{p}_1) \times \mathbf{a}_1| + |(\mathbf{x}_c - \mathbf{p}_2) \times \mathbf{a}_2| + |(\mathbf{x}_c - \mathbf{p}_3) \times \mathbf{a}_3| - |\mathbf{a}_1 \times \mathbf{a}_2| = 0$$

### 3.6.2.2 Tratamiento de colisiones

Una vez conocida la existencia de colisión de la partícula con el triángulo, su tratamiento posterior es exactamente el mismo que el de colisión con un plano, el definido por el triángulo, ya estudiado en la sección anterior.

## 3.7 Estructura de voxels

Originalmente, el *voxel* (contracción del inglés "volume pixel") designó la parte más pequeña distingible de una imagen 3D. Esa noción se ha generalizado rápidamente, perdiendo tanto la pequeñez del volumen como el monopolio del color. Hoy en día una estructura de *voxels* en una región 2D ó 3D finita del espacio está constituida por una discretización de esa región en la que cada una de sus celdas tiene asociado un cierto objeto (color, vector, etc).

En general, y en particular en nuestros casos de simulación gráfica, la estructura de voxels seguirá siendo 3D. En la imagen siguiente vemos un ejemplo muy simple de estructura de voxels constituida por una discretización de un paralelepípedo en  $3 \times 4 \times 2$  celdas, a cada una de las cuales se le ha asociado un vector. Nótese que el espacio de objetos asociado no tiene porqué ser *homogéneo*, es decir, que ciertas celdas de una misma estructura de voxels pueden tener asociados valores numéricos (unas color, otras masa, otras temperatura, etc), otras vectores (unas vectores fuerza, otras vectores velocidad etc), otras matrices (unas de rotación, otras de deformación, otras de tensión, etc).

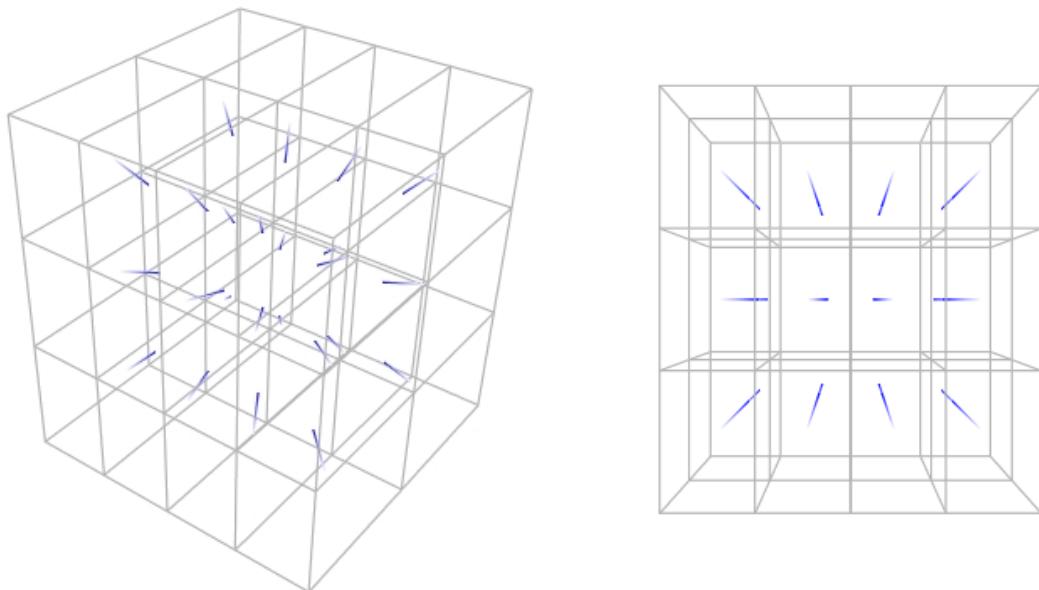


Figura 3.7.1 Estructura de voxels 3D de  $2 \times 3 \times 4$ , donde un voxel mide  $20 \times 30 \times 40$  unidades. Cada voxel contiene un vector fuerza dirigido hacia el centro de la estructura.

Una estructura de voxels queda pués determinada por una red de celdas en una region 2D ó 3D del espacio, una matriz (de segundo o tercer orden) de objetos y una aplicación biyectiva entre las celdas de la red y los elementos de la

matriz. Conviene también asociar, a todo punto exterior a la red, un elemento u objeto nulo,  $e_{nul}$ .

En la figura siguiente ilustramos esta estructura para un espacio 2D. Su extensión al espacio 3D es directa añadiendo un índice más a la matriz y una dimensión más al espacio.

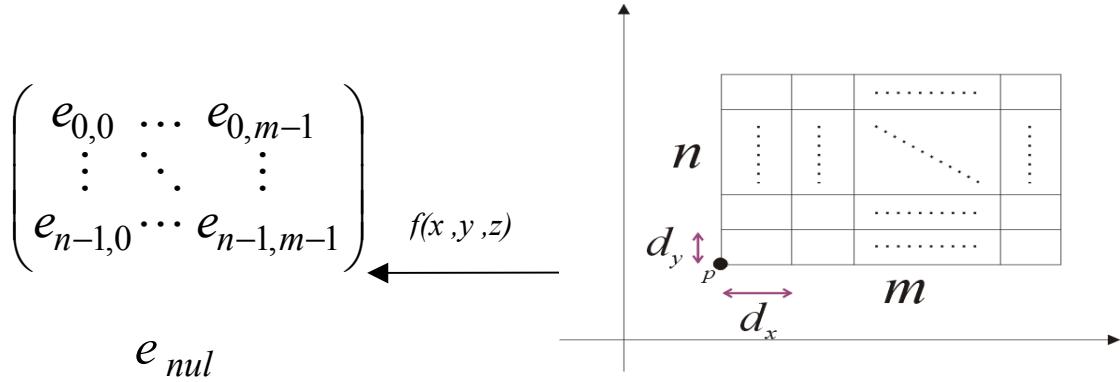


Figura 3.7.2 Estructura de voxels en un espacio 2D

Como se indica en la figura 3.7.2, la red es de  $m \times n$  celdas de tamaño  $d_x \times d_y$ , y  $\mathbf{p}(x_0, y_0)$  es su origen. Un punto  $\mathbf{x}(x, y)$  pertenece a la red si, y solo si,

$$\begin{aligned} 0 \leq x - x_0 &< md_x \quad \text{y} \\ 0 \leq y - y_0 &< nd_y \end{aligned}$$

y pertenece a la celda  $(i, j)$  si, y solo si,

$$\begin{aligned} id_x \leq x - x_0 &< (i + 1)d_x \\ jd_y \leq y - y_0 &< (j + 1)d_y \end{aligned}$$

Denotemos por  $(e_{ij})$  los elementos de la matriz de objetos,  $i = 0, 1, \dots, n-1$ ,  $j = 0, 1, \dots, m-1$ . La aplicación biyectiva  $f$  asocia a todo punto  $\mathbf{x}_{ext}$  exterior a la red el elemento nulo  $e_{nul}$ ,  $e_{nul} = f(\mathbf{x}_{ext})$  y a todo punto  $\mathbf{x}_{int}(x, y)$  interior a ella el elemento  $e_{ij} = f(\mathbf{x}_{int})$  tal que

$$i = E\left[\frac{(x - x_0)}{d_x}\right] \quad \text{y} \quad j = E\left[\frac{(y - y_0)}{d_y}\right]$$

Todos los ingredientes presentados aquí se extienden sin ninguna dificultad al caso 3D.

Veremos la manera de utilizar estas estructura de voxels en los ejemplos de simulaciones. La más destacada es la reconstrucción de objetos 3D a partir de fotografías o radiografías. La idea principal de estos sistemas es la de generar, a partir de las imágenes, un campo de fuerzas convergentes en los contornos de las imágenes y definir por extrapolación el objeto 3D. Para ello, asociamos este campo de fuerzas a una estructura de voxels en la cual insertamos una malla elástica donde cada vértice es una partícula. Estas, que se ven sometidas a las fuerzas de los voxels, tienden a situarse en el contorno definido por el campo de fuerzas. De esta manera queda reconstruido el objetos 3D.

## 4. OpenSIM – Librería C++

### 4.1 Introducción a la librería OpenSIM

OpenSIM es una librería programada en C++, aprovechando las características que ofrece este lenguaje de programación (herencia, polimorfismo en las funciones...). Es multi-plataforma y su propósito es la simulación por ordenador.

En su primera versión, se ha desarrollado su estructura general más un conjunto de clases orientadas a la simulación de sistemas de partículas. Su ampliación podría ser interesante para la unificación de varios campos de la investigación, como podrían ser el proceso de la imagen o la inteligencia artificial en una sola librería. Permitiría programar aplicaciones o hacer pruebas utilizando las prestaciones de todos esos campos de una manera sencilla y donde el investigador se podría dedicar más profundamente a su tarea que a los problemas prácticos de implementación.

Se han definido una serie de primitivas (Objetos básicos, como punto, vector y matriz 3D, coordenadas de textura (2D), plano, línea, etc) a partir de los cuales construimos estructuras más complejas como las de partículas. De momento nos hemos centrado sobre las primitivas relacionadas con los sistemas de partículas y la geometría 3D.

En una primera parte explicaremos brevemente la nomenclatura utilizada en la librería. A continuación detallaremos el diseño de clases que separamos en seis grupos más la clase madre SimObject. Enumeraremos las clases de estos seis grupos más la clase madre, y explicaremos sus funciones más importantes. Finalmente concluiremos, definiendo posibles ampliaciones, unas previstas y otras deseadas.

Para una documentación más detallada sobre OpenSIM, se ha generado automáticamente a partir del código fuente, todo un conjunto de páginas HTML al estilo de la documentación de Java; el lector que desee más información sobre cualquiera de las clases de la librería puede referirse a dicha documentación.

## 4.2 Diseño de clases

En el diseño de la librería se ha intentado ser lo más general posible. Es decir que se ha pensado en estructuras (o clases) lo más cercanas al concepto matemático o físico subyacente y lo más independientes posible. La idea es conseguir de esta manera una facilidad de utilización y, aprovechando las propiedades de la herencia, un reutilización del código fuente, al diseñar cada uno sus propias clases. Podemos agrupar las clases implementadas hasta ahora en seis categorías:

1. Las primitivas
2. Estructuras secundarias
3. Estructuras de datos
4. Estructuras de cálculo
5. Entradas y Salidas a ficheros
6. Salidas a Pantalla.

Todas las clases heredan de la clase madre SimObject. Explicaremos esta clase luego más en detalle, y veremos su utilidad al crear estructuras de datos o de debuggar el programa (control de errores).

Podemos ver la estructura general de OpenSIM en la figura 4.5.1. donde se puede apreciar la separación entre un *núcleo* y las entradas y salidas a otros dispositivos. Esta separación nos permite ampliar el número de dispositivos de entrada/salida fácilmente. Por ejemplo para la visualización de los objetos tenemos varias posibilidades. Las clases implementadas de momento para dicha función son 3:

- OpenGLOutput
- DirectXOutput
- TextOutput

La primera clase, OpenGLOutput, utiliza la librería gráfica OpenGL. Permite crear entornos en 3 dimensiones y visualizar en ellos nuestras simulaciones. Nos ofrece muchas posibilidades al trabajar en 3D. Veremos más tarde cómo dejamos al usuario la libertad de crear sus propias funciones OpenGL para representar algunos de los objetos de la librería. La segunda clase, DirectXOutput (no implementada en su totalidad), es similar a la anterior pero utiliza la librería gráfica de Microsoft DirectX. La última clase, TextOutput, permite visualizar los objetos en la salida estándar de texto. Nos será muy útil al comprobar valores numéricos y controlar nuestro programa.

De forma similar para las entradas y salidas a disco tenemos las clases siguientes:

- Import
- Export
- SQLDataBase

- SQLResult

En las clases Import y Export se implementarán todas las funciones que tengan acceso a disco para cargar y guardar respectivamente datos (u objetos). Para facilitar la utilización de bases de datos SQL se tiene la clase SQLDataBase que permite la conexión a la base de datos y el tratamiento de sentencias SQL, y la clase SQLResult, donde se guarda el resultado de una *query* (sentencia SQL).

El núcleo de la librería esta compuesto por una serie de primitivas, por unas estructuras de datos para poder guardarnos lo objetos, por unas estructuras *segundarias*, formadas a partir de primitivas, y por de unas estructuras (o clases) encargadas de los cálculos. Teniendo en cuenta que hemos orientado esta primera versión de la librería a la simulación de sistemas de partículas, la mayoría de las primitivas están relacionadas con la simulación en el espacio 3D.

Tenemos como primitivas:

- Point3D\_d
- Vector3D\_d
- Matrix3D\_d
- TexPoint2D\_d

Las funciones de estas primitivas son las indicadas por su propio nombre. Son primitivas para trabajar en 3D. Diferenciamos entre punto y vector por la matemática grafica utilizada en los entornos 3D; ésta se caracteriza básicamente por la adición de la coordenada homogénea, que vale 1 para un punto y 0 para un vector. Las matrices 3D son entonces de 4\*4 y no de 3\*3. Eso nos permite aplicar cualquier transformación a un punto o un vector incluyendo la translación (imposible con una matriz de 3\*3). Veremos más en detalle las operaciones permitidas entre estas primitivas. TexPoint2D permite definir un punto de textura (utilizadas en los gráficos 3D). Al ser la textura una superficie el punto queda determinado por 2 parámetros (generalmente denotados s, t).

En las estructuras secundarias veremos todas las otras clases formadas a partir de las primitivas. Las podemos separar en 2 grupos. El primero específico del mundo 3D (donde encontramos clases de geometría 3D y clases específicas de los gráficos 3D por ordenador), y el segundo grupo, específico de los sistemas de partículas en concreto.

## 1. Mundo 3D:

- Line              (geometría)
- Plane             (geometría)
- Triangle         (geometría)
- MeshFace        (gráficos 3D)
- Mesh              (gráficos 3D)

Las primitivas específicas de la geometría nos permitirán trabajar con los objetos matemáticos correspondientes, y serán muy útiles al tratar las colisiones. Las otras dos clases sirven para representar objetos con la propiedades de los gráficos 3D, es decir textura, iluminación, etc.

## 2. Sistemas de partículas:

- Particle
- Spring
- TglSpring
- ParticleSystem
- SpringSystem
- TglSpringSystem

Veremos más en detalle la estructura de estas clases en el apartado [4.3.3] dedicado a las estructuras secundarias.

Las estructuras de datos guardan solamente punteros a SimObject, por lo tanto tendremos la posibilidad de guardar cualquier objeto de la librería. Esto se explicara más en detalle en la sección correspondiente, así como en la sección del objeto madre SimObject. Las clases son:

- List
- Voxel

La clase List es una lista bidireccional. La clase Voxel nos permite crear las estructuras de voxels explicadas en la teoría anterior [3.7].

Por ultimo, tenemos en el núcleo las clases encargadas de los cálculos.

- Solver
- Force

La primera permite resolver las ecuaciones diferenciales ordinarias para funciones del tipo  $y' = f(x, y)$  según los métodos vistos en la sección [3.3] y en particular para el caso de nuestros sistemas de partículas (ecuación diferencial ordinaria de segundo orden). La segunda permite calcular un cierto número de fuerzas para las partículas explicadas en la sección [3.4].

En la figura 4.5.2 se ha detallado la estructura general de librería incluyendo todas la clases implementadas en esta versión.

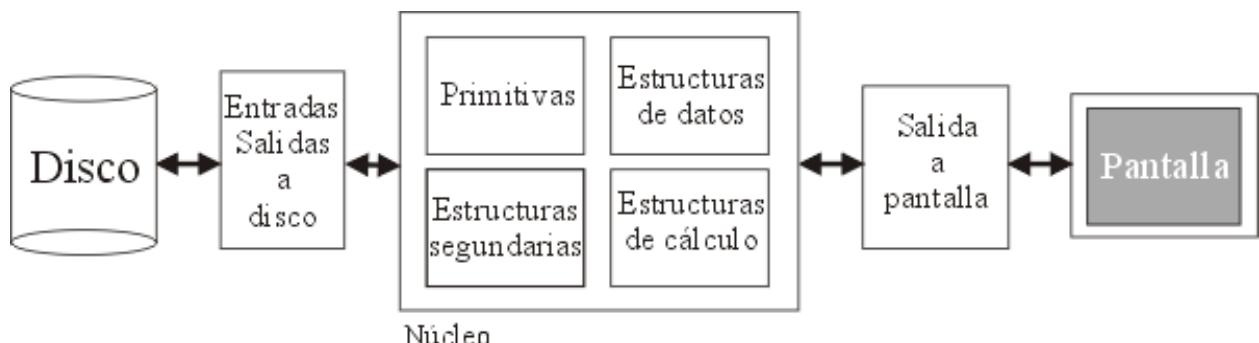


Figura 4.2.1 Diseño de la estructura de la librería de simulación OpenSIM.

# OpenSIM version 1.0

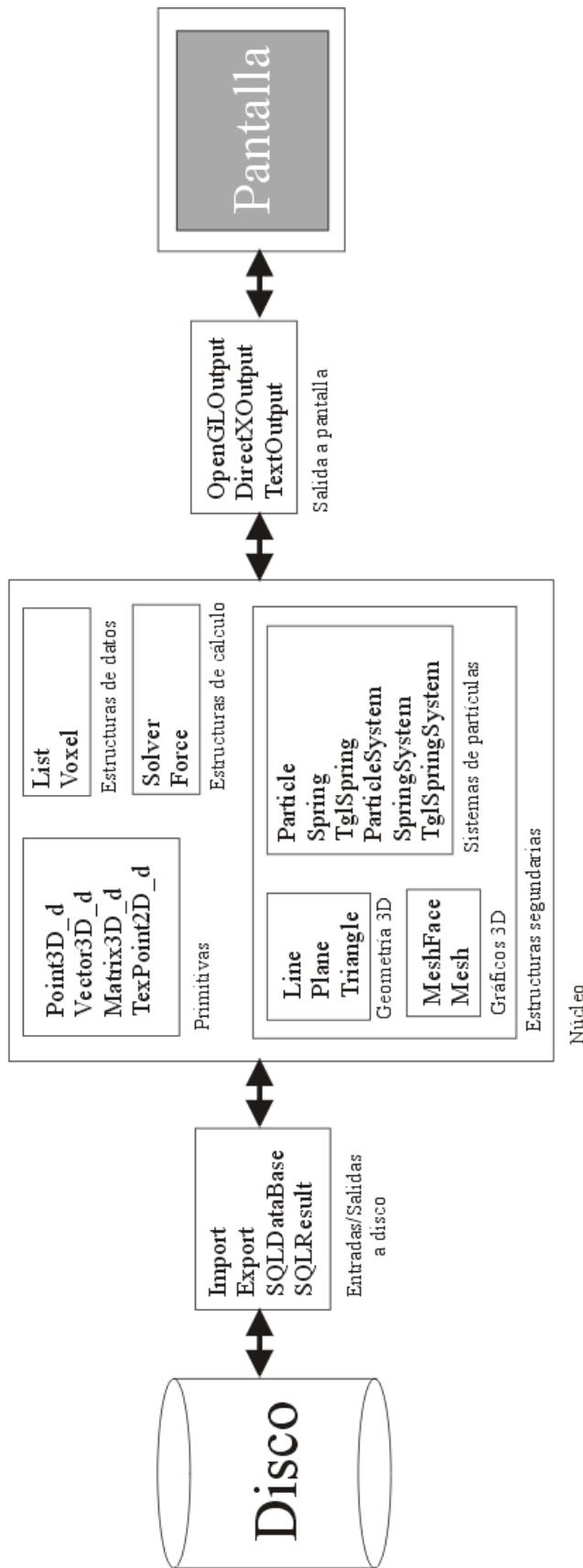


Figura 4.5.2 Diseño detallado de clases de la librería OpenSIM de simulación en su primera versión.

## 4.3 Nomenclatura

Explicaremos brevemente la nomenclatura utilizada en la librería y su estructura básica de ficheros para que el lector pueda aprovechar rápidamente el código de la librería. Se ha seguido generalmente la nomenclatura del conjunto de clases de Java [9] añadiendo algunas reglas propias, es decir:

- El lenguaje utilizado es el inglés, tanto para nombres de clases como para funciones o variables.
- Los nombres de las clases empiezan por una mayúscula  
Ejemplo: `Import`
- Los nombres de funciones o variables empiezan por una minúscula.  
Ejemplo: `int getObjectType()`
- Si un nombre consta de varias partes se empieza cada parte por una mayúscula.  
Ejemplo de nombre de clase: `TglSpringSystem`  
Ejemplo de nombre de función:  
`void setNumOfParticles( int value )`
- Las constantes se escriben exclusivamente en mayúsculas, y además:
  - si consta de varias partes, cada parte va separada por el carácter `_`.
  - si son específicas de una clase, el nombre de la constante empieza por el de la clase (en mayúsculas y sin carácter de separación).Ejemplo: `SIMOBJECT_DEBUG_FILENAME`
  - si son generales, empiezan por `"SIM_"` y se encuentran en el fichero `define.h`Ejemplo: `SIM_X`
- Las extensiones de ficheros (si están en el nombre de la clase o función) se escriben en mayúscula.  
Ejemplo: `bool ASEFileToMesh( ... )`
- El nombre de los objetos que dependen de tipos básicos de datos (`int`, `float`, `double`, `char` ...) se terminan por la primera letra del tipo en cuestión.  
Ejemplo: `Point3D_d (double)`

Estas son las reglas generales de notación que se han seguido, intentando además en todo momento que los nombres de funciones sean lo más explicativos posible. A continuación veremos la estructura básica de ficheros.

Toda clase tiene su fichero nombreClase.h y nombreClase.cpp, por ejemplo la clase Particle se define con los ficheros Particle.h y Particle.cpp. Aparte tenemos los ficheros define.h y OpenSIM.h. En define.h definimos las constantes utilizadas en todas las clases de la librería. En OpenSIM.h tenemos incluidas todas las clases de la librería. De esta manera la inclusión de este fichero basta para la utilización de la librería.

## 4.4 Explicación de las clases

En esta sección se detallan las características y funciones más importantes de las clases de nuestra librería. Empezaremos por la clase padre SimObject y continuaremos con cada una de las categorías de clases definidas en la sección 4.2. Para una explicación más detallada de las clases y funciones, ver la documentación en formato HTML.

### 4.4.1 La clase padre SimObject

Todas las clases heredan de la clase SimObject. Aprovechando las características que nos ofrece el lenguaje de programación C++, podemos crear de esta manera estructuras de datos generales que funcionen para cualquier objeto de la librería. Las estructuras de datos son estructuras de punteros a SimObject. El lenguaje C++ nos permite crear un objeto B que herede de un objeto A con un puntero del tipo A. Podemos crear cualquier objeto de la librería a partir de un puntero a la clase SimObject ya que toda clase hereda de SimObject. En consecuencia las estructuras de datos pueden, en particular, no ser homogéneas. Para saber a qué tipo de objeto corresponde cada puntero SimObject, esta clase tiene el atributo objectType que mediante un entero identifica el tipo de objeto que se ha creado. Por ejemplo si creamos un objeto Particle a partir de un puntero a SimObject, podremos saber que es de tipo Particle mirando el valor de la variable objectType del objeto creado.

Para facilitar la detección de errores, la clase SimObject dispone de la función debug que básicamente escribe un mensaje:

```
void debug(bool onStdout, bool onFile, char* arguments, ...);
```

Los 2 primeros parámetros permiten seleccionar la salida del mensaje, salida estándar i/o fichero. El nombre del fichero de salida está definido por la constante SIMOBJECT\_DEBUG\_FILENAME de la clase SimObject. La salida del mensaje viene adelantada por el nombre del tipo de objeto que envía el mensaje. Por ejemplo si escribimos un mensaje desde un objeto Mesh se verá en la salida :

[ Object Mesh ] → nuestro mensaje

De momento son las 2 funcionalidades básicas implementadas en la clase SimObject : poder crear cualquier objeto a partir de un puntero a SimObject y escribir mensajes en la salida estándar i/o en un fichero de control.

#### 4.4.2 Primitivas

En esta primera versión de la librería, centrada principalmente en la simulación de sistemas de partículas, solo se han implementado 4 primitivas : 3 objetos matemáticos básicos para el tratamiento de simulaciones 3D más el objeto TexPoint\_d:

- Point3D\_d
- Vector3D\_d
- Matrix3D\_d
- TexPoint2D\_d

Los 3 objetos matemáticos son el punto, el vector y la matriz para espacios 3D. En gráficos 3D se le añaden a estos objetos una coordenada más, llamada coordenada homogénea. Esta coordenada vale 1 en el caso del punto y 0 en el del vector. En consecuencia una matriz 3D es de 4\*4 y no de 3\*3. La utilización de esta nueva notación nos permite transformar arbitrariamente un punto o un vector mediante una misma matriz. Podremos definir translaciones, rotaciones y proyecciones con una simple matriz. La notación de matrices es la utilizada en gráficos 3D, en oposición a la notación utilizada en robótica. La figura 4.4.2 ilustra la diferencia entre estas 2 notaciones.

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \quad (a)$$

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} \quad (b)$$

Figura 4.4.2 (a) Notación en gráficos 3D  
(b) Notación en robótica

La translación de un punto en la notación gráfica se hace mediante los coeficientes  $a_{03}$ ,  $a_{13}$ ,  $a_{23}$ , mientras que en la robótica los coeficientes son  $a_{30}$ ,  $a_{31}$ ,  $a_{32}$ .

Desde un punto de vista de programación C++, hemos sobrecargado los operadores que representan las operaciones matemáticas entre estos objetos. Pero esto nos permite escribir nuestras ecuaciones de un modo muy similar al utilizado manualmente sobre papel.

El objeto TexPoint2D\_d es una primitiva para el objeto Mesh (malla 3D). Representa una coordenada de textura. Para mas detalle sobre las mallas 3D ver la sección 4.4.3 donde se explican las estructuras secundarias.

#### 4.4.3 Estructuras secundarias

La categoría de las estructuras secundarias, contiene las clases o estructuras creadas a partir de estructuras primarias o secundarias. En esta primera versión se puede subdividir en 3 grupos : geometría 3D, gráficos 3D, y sistemas de partículas .

El grupo de geometría 3D está formado por las 3 clases siguientes :

- Line
- Plane
- Triangle

Cada clase representa el objeto matemático correspondiente (a su nombre) en un espacio 3D. Por lo tanto tenemos la recta, el plano y el triángulo.

El objeto Line (recta) se define por un punto y un vector (dirección de la recta). Su principal función es la de poder calcular su distancia respecto de un punto.

El objeto Plane (plano) se define por un vector (normal al plano) y un punto. Permite calcular el punto de intersección entre una recta y él. También permite saber si existe colisión con el segmento formado por 2 puntos. Las condiciones de existencia de una colisión están definidas en la sección 3.6.1.1. Además calcula el punto o vector simétrico respecto del mismo.

El objeto Triangle ( triángulo) hereda del objeto Plane ya que un triángulo determina un plano. Se define por tres puntos. Permite funciones análogas a las del plano, pero la condición de existencia de colisión es más restrictiva. Esta condiciona se obtiene en la sección 3.6.2.1.

Gracias con sus funcionalidades, estos objetos serán muy útiles al detectar y tratar colisiones en los sistemas de partículas .

El grupo de gráficos 3D está formado por las 2 clases siguientes :

- MeshFace
- Mesh

Los objetos 3D se definen mediante mallas de triángulos. Para tratar la iluminación, cada triángulo puede tener una normal para todo el triángulo o una para cada vértice. En el primer caso la iluminación es plana (FLAT) mientras que en el segundo la iluminación es suave (SMOOTH). En la figura 4.4.3.1 se ilustra la diferencia entre estas 2 iluminaciones. Por otra parte, un triángulo puede ser relleno mediante una textura (imagen bitmap). Cada triángulo tiene para ese fin 3 coordenadas de textura (objetos de tipo TexPoint2D\_d).

Todas estas propiedades se agrupan en la clase MeshFace (cara de una malla). Para no repetir vértices o coordenadas de textura, la clase MeshFace se referencia a tres vértices y tres coordenadas de textura. La clase Mesh (malla) está definida por una lista de vértices (Point3D\_d), una de coordenadas de textura (TexPoint2D\_d) y una de MeshFace donde cada MeshFace está convenientemente referenciado a puntos y coordenadas de textura de las 2 primeras listas.

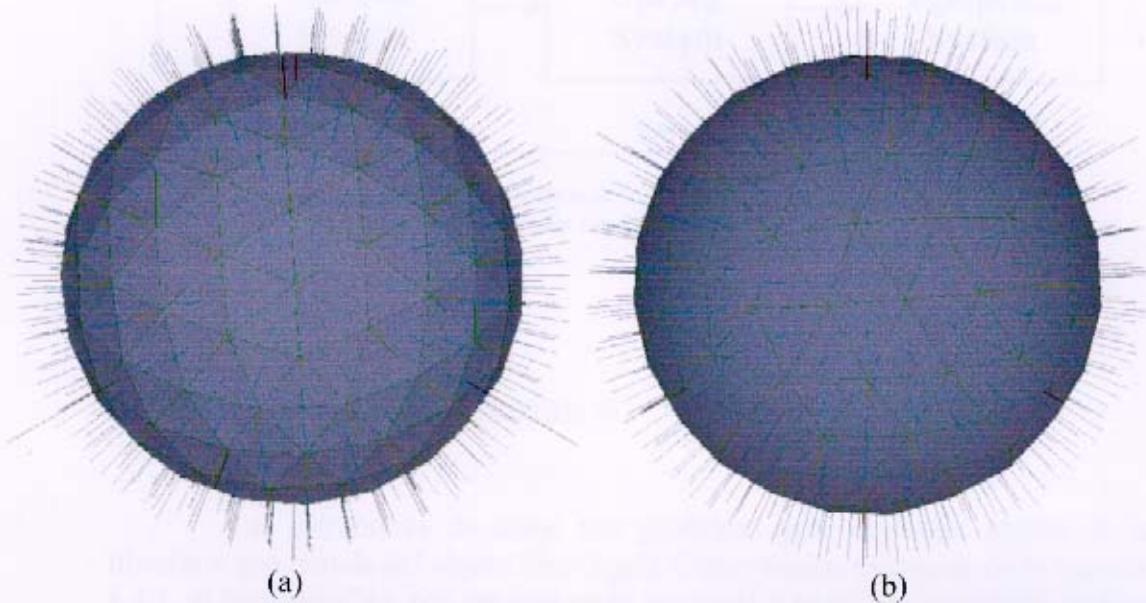


Figura 4.4.3.1

(a) Iluminación FLAT

(b) Iluminación SMOOTH

El grupo de sistemas de partículas está formado por las 6 clases siguientes:

- Particle
- Spring
- TglSpring
- ParticleSystem

- SpringSystem
- TglSpringSystem

La partícula es nuestro elemento básico de simulación. En la sección [3] se explica la teoría computacional de partículas. Un SpringSystem (sistema de partículas) es una lista de partículas. La clase Spring simula un resorte y por lo tanto aplica una fuerza entre 2 partículas. La clase SpringSystem permite trabajar con sistemas de partículas unidas por resortes. Se pueden asignar mallas (objeto Mesh) a sistemas de partículas (con o sin resortes), lo que permite crear mallas que se deforman con el sistema. TglSpring y TglSpringSystem estructuran los sistemas con resortes en triángulos.

En la figura 4.4.3.2 se muestra el esquema que estructura estas seis clases.

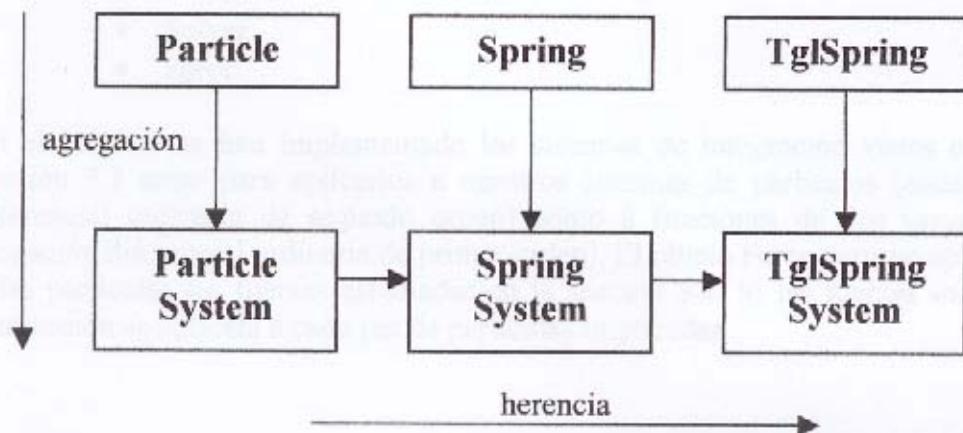


Figura 4.4.3.2 Estructura de las clases relacionadas con los sistemas de partículas.

#### 4.4.4 Estructuras de datos

Las estructuras de datos son genéricas para cualquier objeto de la librería o que herede del objeto SimObject. Como hemos explicado en la sección 4.4.1, el lenguaje C++ nos permite pedir memoria a partir de un puntero de tipo A para todo objeto B que herede de A. Si se quieren utilizar estas estructuras de datos para objetos propios del usuario, hace falta que hereden de SimObject.

Se han programado 2 estructuras de datos:

- List
- Voxel

El Objeto List crea una lista bi-direccional de punteros de tipo SimObject. El objeto Voxel permite generar un mundo de voxels (ver sección 3.7). Se le ha añadido la posibilidad de crear en cada voxel un vector fuerza dirigido hacia el centro de la estructura o hacia el exterior. Veremos en los

ejemplos de simulaciones la posible utilización de estos voxels. En el primer caso conseguiremos efectos de bolas de fuego, mientras que en el segundo caso simularemos explosiones.

#### 4.4.5 Estructuras de cálculo

En la categoría de las estructuras de cálculo reunimos los objetos (o clases) cuya función es la de calcular operaciones matemáticas para la simulación. Los objetos de este grupo son dos:

- Solver
- Force

En el primero se han implementado los sistemas de integración vistos en la sección 3.3 tanto para aplicarlos a nuestros sistemas de partículas (ecuación diferencial ordinaria de segundo orden) como a funciones de dos variables (ecuación diferencial ordinaria de primer orden). El objeto Force permite aplicar a las partículas las fuerzas estudiadas en la sección 3.4. Si las fuerzas son de interacción se aplicará a cada par de partículas implicadas.

#### 4.4.6 Clases de entradas y salidas a disco

Todas las entradas y salidas a disco se centralizan en 2 clases. Estas nos permitirán cargar o grabar objetos como mallas, estructuras de voxels, sistemas de partículas, a partir de ficheros o bases de datos SQL. Las dos clases son:

- Import
- Export

Con la clase Import importaremos (cargaremos) la información del disco y con Export la exportaremos (grabaremos). De momento se han programado funciones que permiten cargar mallas (objetos de tipo Mesh) a partir de ficheros de extensión ASE. Estos ficheros se generan con el programa de animación 3D Studio Max. Se pueden también cargar estructuras de voxels de vectores de fuerza. Para mas información ver la documentación HTML.

Para facilitar el uso de bases de datos SQL se han creado las clases

- SQLDataBase
- SQLResult

La primera, SQLDatabase, conecta con la base de datos y ejecuta sentencias SQL. La segunda, SQLResult, es una estructura (lista bi-direccional de registros) que guarda el resultado de la *query*. La utilización de estas clases es muy simple y será de gran utilidad a la hora de diseñar aplicaciones importantes donde se necesite conexión SQL.

#### 4.4.7 Clases de Salidas a Pantalla

Las salidas a pantalla se controlan a través de tres objetos:

- OpenGLOutput
- DirectXOutput
- TextOutput

Las salidas a pantalla son las que permiten visualizar el resultado de nuestra simulación. Este puede ser gráfico o numérico. En el primer caso utilizaremos los Objetos OpenGLOutput o DirectXOutput. OpenGLOutput utiliza la librería gráfica OpenGL para representar los diferentes objetos (Point3D\_d, Vector3D\_d, Voxel, Mesh, etc) de la librería. DirectXOutput utiliza la librería DirectX de Microsoft. Esta última no es multi-plataforma. Para la salida numérica o “de texto”, se ha programado la clase TextOutput, que escribe por la salida estándar. Utilizaremos esta clase para controlar o detectar posibles errores en la simulación.

## 4.5 Conclusiones sobre la librería

OpenSIM está desarrollado de momento solamente en su primera versión, donde se ha puesto el acento en la simulación de sistemas de partículas. Como hemos visto en la sección 2, los sistemas de partículas abarcan la primera parte de la simulación por ordenador, es decir los objetos deformables. La segunda parte se concentra en la simulación de objetos rígidos.

En la próxima versión de la librería OpenSIM añadiremos, el tratamiento de objetos rígidos. De esta manera conseguiremos un núcleo de simulación lo suficientemente completo como para diseñar situaciones complejas donde se mezclen todo tipo de simulaciones (ver figura 2.1).

En una tercera versión sería interesante añadir funcionalidades de tratamiento de la imagen. Se podrían entonces simular por ejemplo, las reacciones de un robot viendo el escenario 3D a través de una o varias cámaras de video y utilizando algoritmos de tratamiento de imagen para situarse.

### Referencias bibliográficas

En la sección de referencias se han incluido las principales publicaciones científicas sobre OpenSIM. Estas son: «Introducing the OpenSIM library: a framework for the simulation of multi-physics systems», que es la descripción más completa y detallada de la librería. En la sección de referencias adicionales se incluyen otras referencias que describen la implementación de OpenSIM en diferentes tipos de aplicaciones.

En la sección de referencias adicionales se incluye una colección de fuentes complementarias de información sobre OpenSIM.

En la sección de referencias adicionales se incluye una descripción de la librería "OpenSim" que es una implementación de la librería "OpenSim" en Java. Esta descripción es de tipo general y no se detallan las características de la librería. Sin embargo, se incluye una descripción de la librería "OpenSim" en Java y se menciona que es una implementación de la librería "OpenSim". Para más información se recomienda visitar el sitio web de la librería.

## 5.1. Simulación de sistemas físicos básicos

Los sistemas físicos básicos son sistemas que abordan los problemas fundamentales de la mecánica, es decir, la fuerza, la masa, el movimiento, la velocidad, la aceleración, la energía, la temperatura, la densidad, la presión, la velocidad del sonido, la gravedad, etc. Estos sistemas son sistemas que describen la interacción entre los sistemas físicos y su entorno. Los sistemas físicos básicos son sistemas que describen la interacción entre los sistemas físicos y su entorno. Los sistemas físicos básicos son sistemas que describen la interacción entre los sistemas físicos y su entorno.

## 5.1. Comportamiento de las partículas en un campo de fuerzas

Probaremos que una partícula se desplaza siguiendo las fuerzas del sistema. Como es natural, las fuerzas que actúan sobre la partícula son las fuerzas gravitatorias y el efecto de atracción y repulsión de los demás cuerpos. La simulación muestra el resultado obtenido y cómo las interacciones entre las fuerzas impulsan la partícula alrededor del sistema.

### 5. Ejemplos de simulación

Se presentan en esta sección ejemplos de simulaciones. Explicaremos primero el sistema, qué tipo de partículas y fuerzas utilizaremos, sus propiedades y qué fuerzas son externas y cuáles son internas, y después, veremos la manera de programarlos con la librería.

Se ha separado esta sección en tres partes. En la primera estudiaremos sistemas muy simples. Estos nos introducirán al mundo de la simulación y al funcionamiento de la librería. En la segunda parte trataremos sistemas más complejos y también más espectaculares, como es la simulación de ropa. En la última parte hablaremos de la reconstrucción de objetos 3D a partir de un campo de fuerzas.

En la sección 6 de resultados se podrán ver capturas de pantalla correspondientes a la simulación de estos sistemas.

Los ficheros fuentes de estos ejemplos se encuentran en la carpeta "Examples" que acompaña a la librería. Se utiliza la salida OpenGLOutput y la librería Glut [5,15] para la gestión de ventanas. Para facilitar el uso de la librería OpenGL se ha implementado un conjunto de clases denominadas las GLClass. Su utilización está explicada en una documentación HTML. Para más información sobre estas clases ver dicha documentación.

#### 5.1. Simulación de sistemas físicos básicos

Veremos cuatro sistemas básicos, que abarcan los problemas simples de una simulación, es decir utilización de fuerzas físicas, de resortes y de colisiones con planos o triángulos. Los 2 primeros simulan el comportamiento de una partícula sometida a fuerzas eléctricas y gravitacionales. Como ejemplo de la utilización de resortes entre partículas, simularemos el comportamiento de una cuerda elástica sometida a su peso y con una extremidad fija. Por último veremos cómo tratar colisiones con un plano o un triángulo.

### 5.1.1. Comportamiento de electrones en un campo eléctrico

El sistema que programaremos se compone de cuatro partículas del mismo signo (como el movimiento resultante es independiente del signo elegido, por comodidad tomaremos el positivo) alineadas, y tales que las 2 exteriores están fijas. La figura 5.1.1 representa dicha configuración.

Físicamente se puede considerar el sistema como el equivalente al formado por las dos partículas interiores en interacción y sometidas a las fuerzas externas debidas a las partículas bloqueadas. En la programación del ejemplo consideramos las cuatro partículas como nuestro sistema donde dos de ellas están bloqueadas. En ese caso solo tenemos fuerzas internas. La fuerza utilizada es obviamente la fuerza eléctrica vista en la sección 3.4.

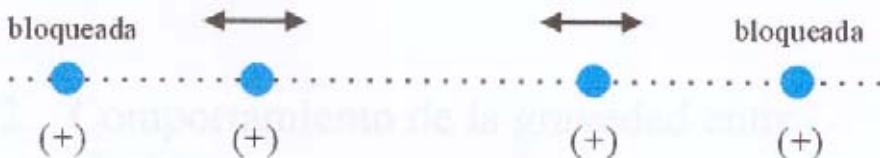


Figura 5.1.1 Sistema eléctrico de partículas

#### Programación con OpenSIM:

Utilizaremos el método Runge-Kutta 4 para integrar nuestras ecuaciones. Por defecto una partícula se inicializa con una carga de valor 1, y se toma como valor de la constante en el cálculo de la fuerza eléctrica 200. Si utilizáramos los valores reales no podríamos apreciar la simulación de nuestro sistema ya que las velocidades de las partículas serían demasiado grandes.

```
//-- variables globales
ParticleSystem ps; // Nuestro sistema de partículas
Solver solver; // Solver numérico
OpenGLOutput openGLOutput; // Salida a pantalla OpenGL

// -- Función que calcula la fuerza
void calculateElectricForce( ParticleSystem *ps, double t)
{
    // reiniciamos el la fuerza acumulada en las partículas
    ps->resetForce();

    // Método estático de la clase Force
    Force::electric( *ps, 200 );
}

// -- En la función de inicialización del programa

// inicializamos el numero de partículas de nuestro sistema
ps.setNumParticles(4);
// Posicionamos cada partícula
ps.getParticleAt(0).getPosition().set(-10,0,0);
ps.getParticleAt(1).getPosition().set(-5,0,0);
```

```

ps.getParticleAt(2).getPosition().set( 5,0,0);
ps.getParticleAt(3).getPosition().set( 10,0,0);
// Bloqueamos las partículas de los extremos
ps.getParticleAt(0).block();
ps.getParticleAt(3).block();
// -- En el bucle principal del programa

// Calculamos la nuevas posiciones del sistema con el solver
// Utilizamos aquí el método Runge-Kutta 4, ver sección [3.3.5]

solver.rungeKutta4( ps, calculateElectricForce );

// Representamos gráficamente el sistema utilizando la salida
// OpenGL. A demás de representar la posición de la partículas
// (primer "true") dibujamos su velocidad (segundo "true") y su
// fuerza (tercer "true")

openGLOutput.render( ps, true, true ,true );

```

Podemos apreciar la simplicidad de programación que nos ofrece la librería OpenSIM. Para los resultados del ejemplo ver la sección 6.

### 5.1.2. Comportamiento de la gravedad entre planetas

El sistema estará constituido por 2 partículas de masa  $m_1 = 100$  y  $m_2 = 1$ . En las condiciones iniciales del sistema, la partícula de masa  $m_1$  tendrá velocidad inicial cero mientras que la partícula de masa  $m_2$  tendrá una velocidad perpendicular a la dirección formada por las 2 partículas. En función de esta velocidad, dicha partícula se satelizará describiendo una trayectoria elipsoidal más o menos pronunciada, caerá sobre la partícula  $m_1$  describiendo una parábola, o se escapará de forma hiperbólica. Los valores de las constantes no corresponden a la realidad por las mismas razones que en el ejemplo anterior.

En este sistema solo tenemos la fuerza (interna) gravitacional entre las dos partículas.

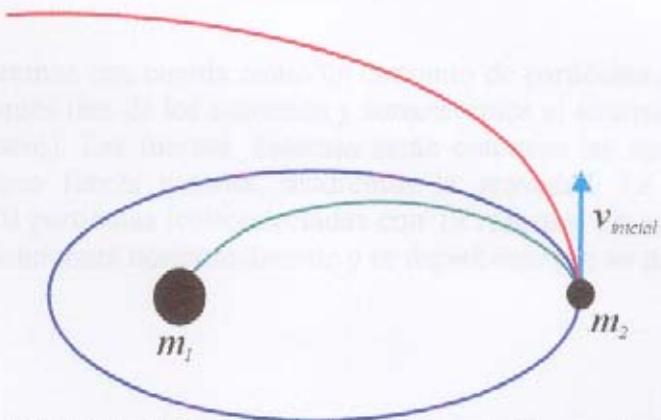


Figura 5.1.2 Trayectorias posibles de la partícula  $m_2$  según su velocidad inicial

### Programación con OpenSIM:

```
//-- variables globales

ParticleSystem ps;           // Nuestro sistema de partículas
Solver solver;              // Solver numérico
OpenGLOutput openGLOutput;   // Salida a pantalla OpenGL

// -- Función que calcula la fuerza

void calculateGravitationalForce( ParticleSystem *ps, double t)
{
    // reiniciamos el la fuerza acumulada en las partículas
    ps->resetForce();

    // Método estático de la clase Force
    Force::gravitacional( *ps, 1 );
}

// -- En la función de inicialización del programa

// Creamos nuestras 2 partículas de nuestro sistema
ps.setNumOfParticles(2);

// inicializamos la primera partícula
ps.getParticleAt(0).getPosition().set( 0,0,0 );
ps.getParticleAt(0).setMass(100);

// inicializamos la segunda partícula
ps.getParticleAt(1).getPosition().set(10,0,0);
ps.getParticleAt(1).getSpeed().set(0,0,-2.5);
ps.getParticleAt(1).setMass(1);

// -- En el bucle principal del programa

// Calculamos la nuevas posiciones del sistema con el solver
solver.rungeKutta4(ps, calculateElectricForce);
// Los mismos parámetros que en el ejemplo anterior
openGLOutput.render(ps, true, true ,true );
```

Para los resultados de este ejemplo, ver la sección 6.

### 5.1.3. Simulación de una cuerda elástica

Simularemos una cuerda como un conjunto de partículas conectadas con resortes. Fijaremos uno de los extremos y someteremos el sistema a la fuerza de la gravedad (peso). Las fuerzas internas serán entonces las aplicadas por los muelles y, como fuerza externa, tendremos la gravedad. La simulación la haremos con 20 partículas interconectadas con 19 resortes. En su estado inicial la cuerda se posicionará horizontalmente y se dejará caer por su propio peso.

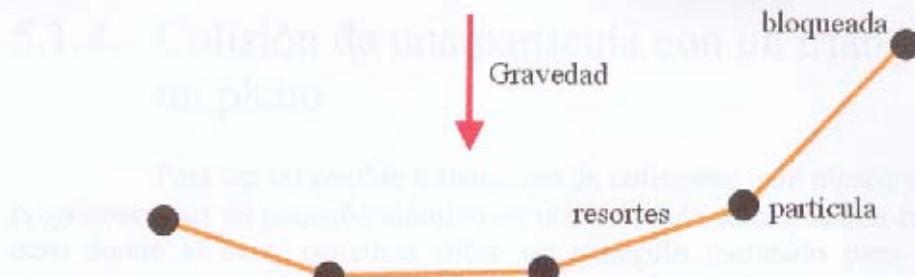


Figura 5.1.3 Simulación de una cuerda.

#### Programación con OpenSIM:

```

//--- variables OpenSIM

SpringSystem ss;           // Nuestro Sistema de con resortes
Solver solver;             // Motor numérico
OpenGLOutput openGLOutput; // salida de tipo OpenGL
Vector3D_d gravity(0,-9,0); // fuerza de la gravedad (peso)

// -- Función que calcula la fuerza

void calculateForce(ParticleSystem *ps, double t)
{
    ps->resetForce();          // reinicializamos el sistema

    // Calculamos la fuerza interna del sistema. Hemos de hacer un
    // conversión a SpringSystem ya que ParticleSystem no pose tal
    // función.

    ((SpringSystem*)ps)->calculateForce();

    // Calculamos la fuerza del peso (fuerza externa)
    Force::constant(*ps, gravity);
}

// -- En la función de inicialización del programa

// inicializamos el numero de partículas que constituirán la cuerda
ss.setNumParticles(20);

// Inicializamos el numero de resortes (un menos que partículas)
ss.setNumSprings(ss.getNumParticles() - 1);

for(int i = 0 ; i < ss.getNumParticles() - 1 ; i++)
{
    ss.getSpringAt(i).setParticles( ss.getParticleAt(i),
                                    ss.getParticleAt(i+1));
    ss.getSpringAt(i).setConstants(10,20,2);
}
for( i = 0 ; i < ss.getNumParticles() ; i++)
{
    ss.getParticleAt(i).getPosition().set(0, 100 , i*10);
}
ss.getParticleAt(0).block(); // Bloqueamos la primera partícula

// -- En el bucle principal del programa

solver.rungeKutta4( ss, calculateForce);
openGLOutput.render( ss, true, true, false, true);

```

Para el resultado de la simulación, ver la sección 6.

### 5.1.4. Colisión de una partícula con un triángulo y un plano

Para ver un posible tratamiento de colisiones con planos y triángulos programaremos un pequeño ejemplo de una partícula sometida a la fuerza de su peso donde al caer, colisiona sobre un triángulo inclinado para finalmente aterrizar sobre un plano horizontal. La única fuerza del sistema, el peso es externa. Trataremos las colisiones con los procedimientos vistos en la sección 3.6. Primero estudiaremos la posible existencia de la colisión y en caso afirmativo, la trataremos con una reflexión amortiguada respecto del plano tanto en la posición de la partícula como en su velocidad. En la figura 5.1.4 se puede ver la configuración del sistema.

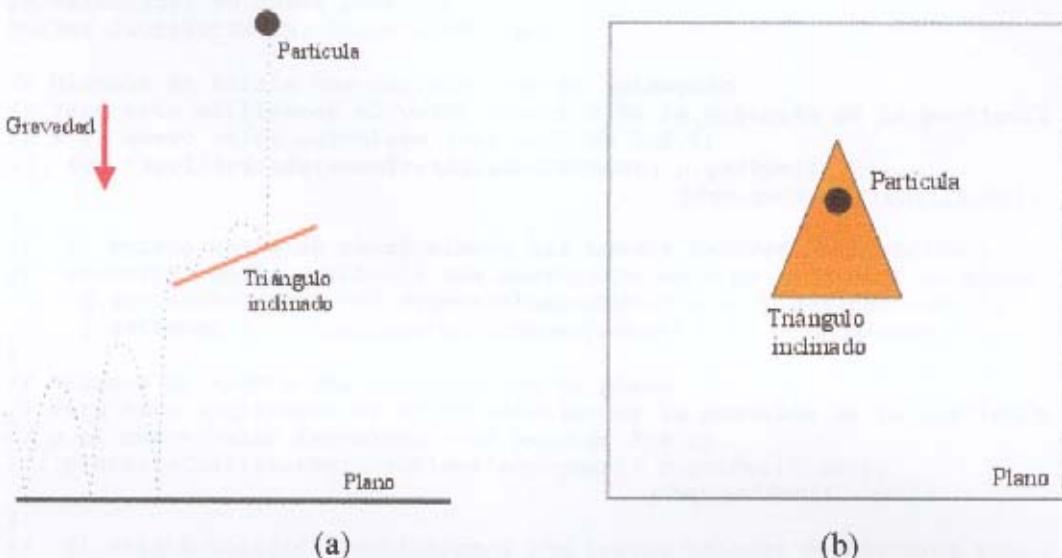


Figura 5.1.4      Configuración del sistema  
(a) vista frontal  
(b) vista desde arriba

#### Programación con OpenSIM:

```
// Variable globales de OpenSIM

Particle      p;                      // Nuestra Particula
Triangle      tgl;                    // el triangulo
Plane         plane;                  // el plano
Vector3D_d    gravity(0,-9,0);        // La fuerza de gravedad
Solver         solver;                 // integrador numérico
OpenGLOutput  openGLOutput;          // Salida OpenGL

// -- Función que calcula la fuerza

void calculateForce(Particle *p, double t)
{
    // reinicializamos el acumulador de la particula
    p->resetForce();
    // Aplicamos la gravedad
    Force::constant(*p, gravity);
```

```

}

// -- En la función de inicialización del programa

Point3D_d p0( 0, 60, -10); // primer punto del triangulo
Point3D_d p1( 20, 55, 40); // segundo punto del triangulo
Point3D_d p2(-20, 55, 40); // tercer punto del triangulo
Point3D_d p4( 0,0,0); // punto del plano
Vector3D_d n(0,1,0); // Vector normal del plano

tgl.set(p0, p1, p2); // inicializamos el triangulo
plane.set( n, p4); // inicializamos el plano

p.getPosition().set(0,130,0); // inicializamos la particula

// -- En el bucle principal del programa

// guardamos el valor de la particular antes de calcular
Particle pTmp = p;
// calculamos su nueva posición
solver.rungeKutta4(p, calculateForce);

// Miramos Si existe una colisión con el triangulo
// Para esto utilizamos el valor anterior de la posición de la particula
// y el nuevo valor calculado (ver sección 3.6.2)
if( tgl.isCollisionBetweenTriangleAndSegment( p.getPosition(),
                                              pTmp.getPosition(),0.01))
{
    // Si existe colisión recalcularmos los nuevos valores de posición y
    // velocidad de la particula con amortiguamiento en la normal al plano
    p.getPosition() = tgl.symmetricDampedPoint(0.4, p.getPosition());
    p.getSpeed() = tgl.symmetricDampedVector( 0.4, p.getSpeed() );
}

// Miramos Si existe una colisión con el plano
// Para esto utilizamos el valor anterior de la posición de la particula
// y el nuevo valor calculado (ver sección 3.6.1)
if( plane.isCollisionBetweenPlaneAndSegment( p.getPosition(),
                                              pTmp.getPosition()))
{
    // Si existe colisión recalcularmos los nuevos valores de posición y
    // velocidad de la particula con amortiguamiento en la normal al plano
    p.getPosition() = plane.symmetricDampedPoint( 0.8, p.getPosition());
    p.getSpeed() = plane.symmetricDampedVector( 0.8, p.getSpeed() );
}

```

Para los resultados de este ejemplo, ver la sección 6.

## 5.2. Modelado de sistemas más complejos

Las simulaciones que se describen a continuación son algo más elaboradas que las anteriores y utilizan funcionalidades más avanzadas de los gráficos 3D y de la librería OpenSIM, como por ejemplo la aplicación de texturas o la utilización de ficheros externos para configurar el sistema. Esto permite obtener resultados mucho más espectaculares. En algunos ejemplos como en la simulación de bolas de fuego, contrastaremos los resultados obtenidos con y sin texturas. Siendo la simulación “física” la misma, la diferencia visual es enorme. Es importante tener ésto en cuenta ya que en muchas simulaciones lo importante es la *apariencia* de realidad y no la *realidad* de la descripción. Para obtener buenas simulaciones tendremos que documentarnos sobre las posibilidades de la librería de salida utilizada, de momento OpenGL o DirectX. En nuestro caso hemos escogido para estos ejemplos la librería OpenGL por su portabilidad entre las diferentes plataformas.

### 5.2.1. Simulación de agua

En este ejemplo no simularemos el agua como volumen sino como superficie, que consideraremos como objeto deformable. La malla utilizada para crearlo se puede ver en la figura 5.2.1 donde también mostramos el resultado visual de dicha malla con textura. Cada arista será un resorte cuyo tamaño en reposo es el mismo que el de la arista. Por lo tanto si fijamos las partículas del borde de nuestra malla, el sistema tiende a quedarse plano. Para la simulación estiraremos una partícula del centro para perturbar la superficie.

En OpenSIM hay una clara diferencia entre la malla representable (con propiedades de textura, de iluminación , etc) y el sistema de partículas en si. Para unir estos dos objetos se puede asignar una malla al sistema de partículas haciendo corresponder los vértices de la malla a las posiciones de las partículas.

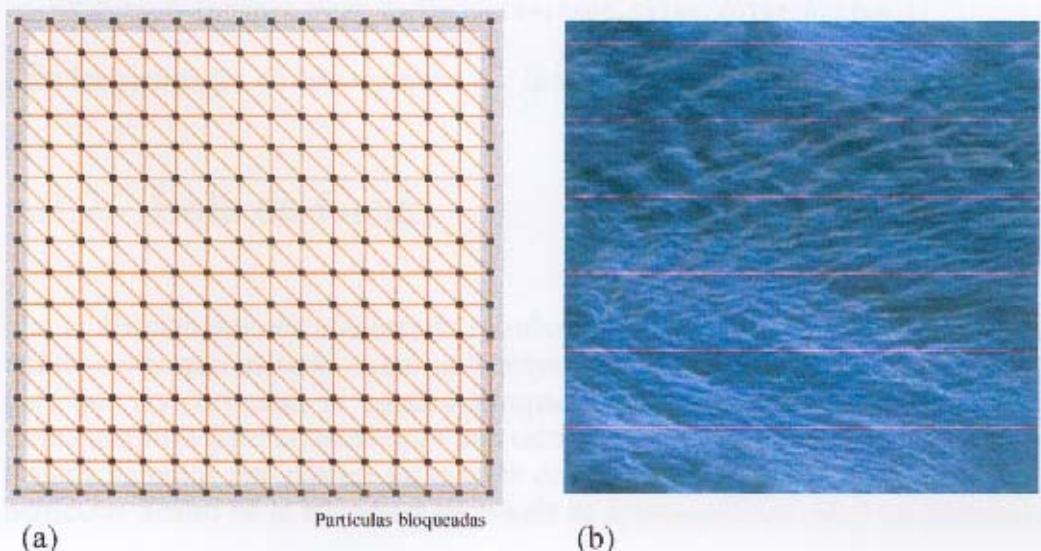


Figura 5.2.1 (a) Sistema de partículas con resortes.

(b) Malla asignada al sistema con la textura de agua.

en la sección 6 para el desarrollo de este ejercicio por los mismos.

### Programación con OpenSIM:

```
// Variable globales de OpenSIM

Mesh           mesh;          // Malla de la superficie del agua
SpringSystem   ss;            // Nuestro Sistema con resortes
Solver         solver;        // El solver numérico
OpenGLOutput   openGLOutput; // Salida OpenGL

// -- Función que calcula la fuerza

void calculateForce( ParticleSystem* ps, double t )
{
    ((SpringSystem*)ps)->resetForce();
    ((SpringSystem*)ps)->calculateForce();
}

// -- En la función de inicialización del programa

// Cargamos un fichero ASE creado con 3D Studio Max para la superficie
// del agua
Import::ASEFileToMesh("./Meshes/plane.ase", mesh);
// Creamos nuestro sistema de partículas y resortes a partir de la malla
ss.createFromMesh(mesh, SPRINGSYSTEM_USE_MESH_SIZE, 10,0);
// Asignamos la malla al sistema
mesh.assignToParticleSystem(ss);

// Bloqueamos todas las partículas de la periferia
for(int i = 0 ; i < 16 ; i++ )
{
    ss.getParticleAt(i).block();
    ss.getParticleAt(i*16).block();
    ss.getParticleAt(i*16 + 15).block();
    ss.getParticleAt(i + 240).block();
}

// Estiramos una partícula del centro del sistema
ss.getParticleAt(135).getPosition().setValueAt(SIM_Y,30);

// -- En el bucle principal del programa

// Integramos el sistema
solver.rungeKutta4(ss, calculateForce);
// Dibujamos la malla si (se puede poner una textura)
openGLOutput.render( mesh, false, false, true, false, 0, true );
```

Para los resultados de este ejemplo, ver la sección 6.

### 5.2.2. Bola de Fuego

Existen muchas maneras de simular bolas de fuego, en realidad cada uno utiliza su propio método. Aquí utilizaremos un campo de fuerzas concéntrico para que las partículas se queden agrupadas. Se podría definir esta fuerza de forma analítica pero emplearemos una estructura de voxels para este fin. De esta manera veremos una posible utilización de los voxels. Situaremos el sistema de partículas dentro de la estructura de voxels de forma esférica. Aquí las partículas

no interactúan entre ellas, solo tenemos la fuerza ejercida por los voxels (fuerza externa). En la figura 5.2.2.1 se ilustra el sistema.

Con el sistema definido conseguimos mover unas partículas alrededor de un centro, pero la similitud del sistema con una bola de fuego depende de la manera de representar las partículas. Se utilizarán las funciones de “blending” (transparencia) que ofrece la librería OpenGL para conseguir el efecto deseado y aplicaremos una textura como en la figura 5.2.2.2 para cada partícula. Por lo tanto cada partícula se representa mediante un cuadrado con la textura anterior.

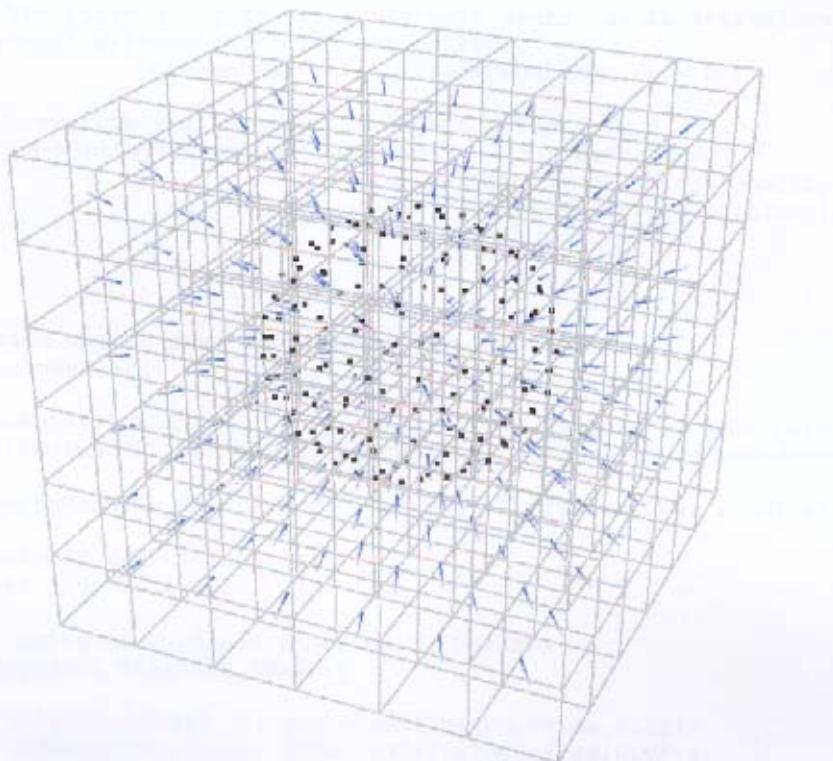


Figura 5.2.2.1 Sistema utilizado para simular una la bola de fuego.  
Utilizamos una estructura de voxels.

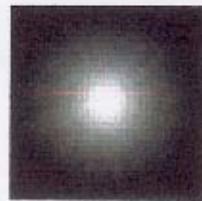


Figura 5.2.2.2 Textura utilizada para simular el fuego.

#### Programación con OpenSIM:

```
//-- variables OpenSIM  
  
Voxel voxel; // estructura de voxels  
ParticleSystem ps; // nuestro sistema de partículas  
Solver solver; // integrador numérico  
OpenGLOutput openGLOutput; // salida OpenGL
```

```

float Size = 30; // tamaño de una partícula
float particleColor[] = {1,0.8,1,1}; // Color de la partícula

// -- Función que calcula la fuerza
void calculateForce(ParticleSystem *ps, double t)
{
    // borramos el acumulador de fuerzas
    ps->resetForce();

    // para cada partícula miramos la fuerza que le corresponde en la
    // estructura de voxels
    for( int i = 0 ; i < ps->getNumParticles() ; i++ )
    {
        // Verificamos que la partícula este dentro de la estructura
        if(voxel.getSimObjectPointerAtPosition(
            ps->getParticleAt(i).getPosition())!= 0 )
        {
            // Convertimos el puntero SimObject a Vector3D_d
            ps->getParticleAt(i).getForce() += *((Vector3D_d*)
                voxel.getSimObjectPointerAtPosition(
                ps->getParticleAt(i).getPosition()) );
        }
    }
}

// -- Función OpenGL para representar las partículas
void particleRender()
{
    glEnable(GL_BLEND); //activa la transparencia
    glBlendFunc(GL_SRC_ALPHA,GL_ONE); //función de transparencia

    glDepthMask(GL_FALSE); //desactivar z-buffer

    glColor4fv(particleColor);
    float side = Size/2;

    // polígono cuadrado donde va la textura
    glBegin(GL_TRIANGLE_STRIP);

    glTexCoord2d(1,1); glVertex3f(+side,+side,0.0f);
    glTexCoord2d(0,1); glVertex3f(-side,+side,0.0f);
    glTexCoord2d(1,0); glVertex3f(+side,-side,0.0f);
    glTexCoord2d(0,0); glVertex3f(-side,-side,0.0f);
    glEnd();

    glDepthMask(GL_TRUE); //activar z-buffer
    glDisable(GL_BLEND); //desactiva transparencia
}

// -- En la función de inicialización del programa

Mesh mesh;
// cargamos una geoesfera en formato ASE
Import::ASEFileToMesh("./Meshes/geoesfera.ase", mesh );
// Creamos el sistema de partículas a partir de la geoesfera
ps.createFromMesh(mesh);

voxel.setSize(10,10,10); // numero de voxels
voxel.setVoxelSize(20,20,20); // tamaño de cada voxel
voxel.setOrigen(-100,-100,-100); // origen de la estructura
voxel.createCentricVectors( 10 );// crear campo de vectores concéntricos

// -- En el bucle principal del programa

// Método de integración
solver.euler(ps, calculateForce );

```

```
// Aquí añadimos la textura ( ver el código entero del ejemplo)
openGLOutput.render(ps, particleRender );
```

Para los resultados de este ejemplo, ver la sección 6.

### 5.2.3. Explosiones

Una explosión, considerada en su aspecto visual y no en el físico, se manifiesta por una gran aceleración momentánea (en todas direcciones) de los puntos de un sistema. Esta aceleración la podemos aplicar, bien en un solo instante, bien durante un intervalo de tiempo.

Para simular el primer caso, de aceleración instantánea, construiremos un sistema de partículas tal que, en el momento de la explosión, se posicionarán todas las partículas en un mismo sitio y se aumentarán sus velocidades considerablemente en una dirección aleatoria.

Para simular el segundo caso, de aceleración continua, se puede definir un estado de explosión mediante una función de fuerza que genere una aceleración. Una vez transcurrido el intervalo de tiempo, pasaremos a otro estado donde la función dejará de aumentar la aceleración.

En nuestro ejemplo hemos simulado el primer tipo de explosiones. La función explotar se acciona en el momento deseado por el usuario mediante un evento de teclado.

#### Programación con OpenSIM:

```
///-- variables OpenSIM

ParticleSystem      ps;
Solver             solver;
OpenGLOutput       openGLOutput;

// -- Función que calcula la fuerza

void calculateForce(ParticleSystem *ps, double t)
{
    // no hace nada
}

// -- Función de explotar

void explode(ParticleSystem &ps, int amplitud)
{
    float tmp = (float)(amplitud)/2;
    for (int i = 0 ; i < ps.getNumParticles() ; i++)
    {
        ps.getParticleAt(i).getPosition().set(0,0,0);
        ps.getParticleAt(i).getSpeed().set( rand()%amplitud-tmp,
                                            rand()%amplitud - tmp,
                                            rand()%amplitud - tmp);
    }
}

// -- En la función de inicialización del programa

ps.setNumParticles(500);

// -- En el bucle principal del programa
```

```
solver.euler(ps, calculateForce );
```

Para los resultados de este ejemplo, ver la sección 6.

### 5.2.4. Fuegos artificiales (o volcán)

Simularemos ahora un fuego continuo como el de una bengala o el de un volcán. Para ello añadiremos a las partículas un tiempo de vida de duración aleatoria, pasado el cual volveremos a inicializar las posiciones y velocidades de las partículas. Conseguiremos de esta forma un flujo continuo de partículas en las direcciones determinadas por las condiciones de inicialización de las partículas. Añadimos al sistema la fuerza de la gravedad. Se tomará como tiempo de vida un valor entre 0 y 1. En 0 la partícula "muere" (desaparece). Guardamos este valor en la variable C (carga) de la clase Particle ya que en nuestro sistema no la utilizamos.

#### Programación con OpenSIM:

```
//-- variables OpenSIM
Vector3D_d gravity(0,-9,0);
ParticleSystem ps;
Solver solver;
OpenGLOutput openGLOutput;

// -- Función que calcula la fuerza

void calculateForce(Particle *p, double t)
{
    p->resetForce();
    Force::constant(*p,gravity);
}

// -- Función de inicialización de una partícula

void initialize(Particle &p)
{
    // inicializamos a velocidad aleatoriamente dentro de un cono
    // que se abre en dirección del eje Y
    p.getSpeed().set(rand()%10 - 5,rand()%40,rand()%10-5);
    // Posicionamos la partícula en el origen
    p.getPosition().set(0,0,0);
    // inicializamos su tiempo de vida
    p.setCharge((float)rand()/(float)RAND_MAX);
}

// -- En la función de inicialización del programa

ps.setNumOfParticles(500);      // creamos 500 partículas
for(int i = 0 ; i < ps.getNumOfParticles() ; i++)
{
    initialize(ps.getParticleAt(i)); // Las inicializamos
}

// -- En el bucle principal del programa
for(int i = 0 ; i < ps.getNumOfParticles() ; i++)
{
    // calculamos la nueva posición de la partícula
    solver.euler(ps.getParticleAt(i), calculateForce);
    // disminuimos su tiempo de vida
    ps.getParticleAt(i).setCharge(ps.getParticleAt(i).getCharge()-0.01);
    // Si la partícula muere la inicializamos de nuevo
    if( ps.getParticleAt(i).getCharge() < 0 )
    {
        initialize(ps.getParticleAt(i));
    }
}
```

Para los resultados de este ejemplo, ver la sección 6.

### 5.2.5. Simulación de ropa

Para simular una tela cuadrada utilizaremos la misma malla que en el ejemplo de la simulación de la superficie del agua (ver sección 5.2.1 e imagen 5.2.1). Simplemente le aplicaremos otras fuerzas externas y otras constantes para las fuerzas internas(resortes).

Una tela, al tener una gran superficie respecto de su peso, se ve muy frenada por el aire, y en todas direcciones. Para simularla, consideraremos el aire como un medio viscoso y aplicaremos en consecuencia una fuerza de viscosidad (ver sección 3.4.1), que se opondrá al movimiento de las partículas de la tela.

Las constantes de los resortes de nuestro sistema se fijarán por tanteo, probando valores hasta encontrar unos que satisfagan el efecto deseado de nuestra simulación.

El ejemplo consiste en una tela estirada horizontalmente y sujetada por un borde. Dejaremos caer la tela por la fuerza de su peso frenando el movimiento de las partículas con una cierta fuerza de viscosidad. Por lo tanto nuestro sistema se verá afectado por 2 fuerzas externas (gravedad y viscosidad ) y por las fuerzas de los resortes(internas).

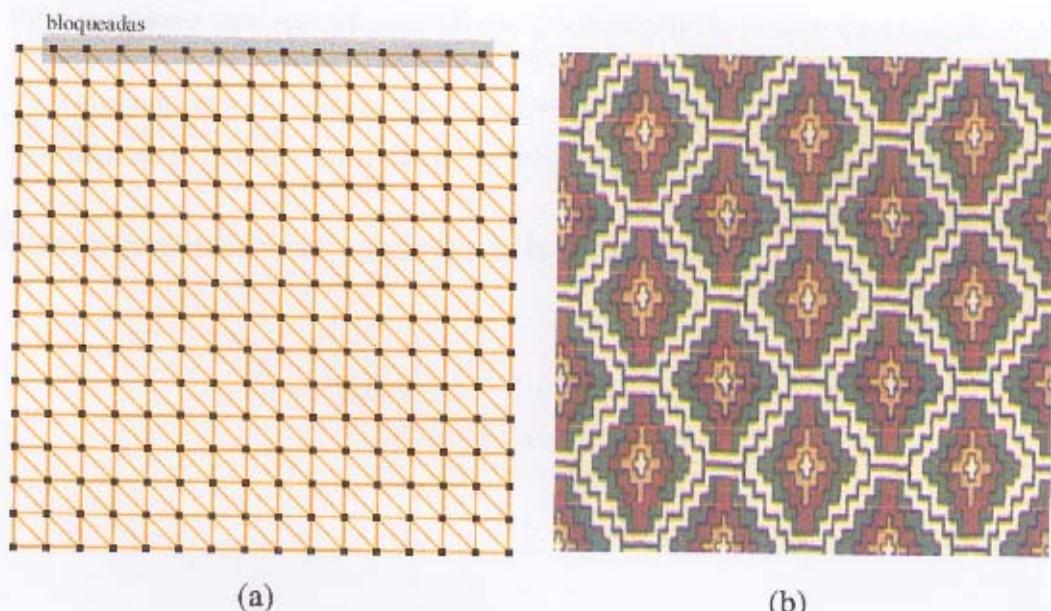


Figura 5.2.5 (a) Sistema de partículas con resortes  
(b) Malla con textura

#### Programación con OpenSIM:

```
///-- variables OpenSIM

Vector3D_d    gravity(0,-9,0);           // Vector gravedad

Mesh          mesh;                      // malla que representa la tela
SpringSystem ss;                      // el sistema con resortes
Solver        solver;                   // integrador
```

```

OpenGLOutput openGLOutput;           // Salida a pantalla OpenGL

// -- Función que calcula la fuerza

void calculateForce( ParticleSystem* ps, double t )
{
    // reiniciamos las fuerzas
    ((SpringSystem*)ps)->resetForce();
    // calculamos las fuerzas internas (resortes)
    ((SpringSystem*)ps)->calculateForce();

    // Calculamos las fuerzas externas (peso y viscosidad del aire)
    Force::constant(*ps, gravity);
    Force::viscosity(*ps, 0.1);
}

// -- En la función de inicialización del programa

// cargamos la malla para la tela
Import::ASEFileToMesh("../Meshes/planoRopa.ase", mesh);
// Creamos el sistema de partículas y resortes a partir de la malla
ss.createFromMesh(mesh, SPRINGSYSTEM_USE_MESH_SIZE, 30, 2);
// Asignamos la malla al sistema de partículas para que se mueva con el
mesh.assignToParticleSystem(ss);

// Bloqueamos un extremo de la tela
for(int i = 1 ; i < 15 ; i++)
{
    ss.getParticleAt(i).block();
}
// -- En el bucle principal del programa

// Calculamos las nuevas posiciones y velocidades de las partículas
solver.rungeKutta4(ss, calculateForce);

// Representamos la malla. Si le ponemos textura el efecto será mucho
// más espectacular
openGLOutput.render( mesh, false, false, true, false, 0, true );

```

Para los resultados de este ejemplo, ver la sección 6.

### 5.3. Reconstrucción de objetos 3D a partir de un campo de fuerzas

La reconstrucción de objetos 3D a partir de captaciones 2D es un tema de investigación muy actual, dado el interés que dicha reconstrucción puede tener en muchos ámbitos, como el de la medicina por ejemplo. A partir de imágenes o captaciones 2D (escáneres, radiografías, etc.) se podrá reconstruir, mediante un procesado de la imagen y de nuestros sistemas de partículas, el objeto observado. Podremos por ejemplo recuperar un corazón en 3D a partir de imágenes médicas, el cual podrá ser observado por el médico en todas direcciones para ver su estado. Ello facilitará el trabajo del medico, ya que sin este procedimiento tendría que imaginarse la forma del corazón a partir del conjunto de las captaciones, lo cual puede ser muy difícil. Entre otras aplicaciones posibles, está la de crear un objeto 3D a partir de un conjunto de fotografías de él. La función de una tal aplicación es la misma que la de un escáner 3D.

Primero explicaremos el proceso teórico para la reconstrucción de objetos 3D, es decir la idea principal, el tipo de procesado de imagen que necesitamos, etc.

Luego veremos un ejemplo sencillo programado con la librería OpenSIM. No se ha utilizado ningún tipo de procesado de la imagen para generar el campo de fuerzas, sino que lo hemos creado a partir de una función de energía. Esto simplifica la programación del ejemplo, y nos evita entrar en el tema de procesado que no cae dentro de los objetivos de esta versión del proyecto OpenSIM.

La idea principal de estos sistemas de reconstrucción 3D, se puede resumir en los pasos siguientes:

1. Recuperar el contorno del objeto a partir de la imagen. Se puede utilizar cualquier tipo de algoritmo destinado a este fin. Uno que funciona bien es el algoritmo de Canny de detección de contornos.
2. Aplicar un procesado a la imagen del contorno para crear un campo gradiente de fuerzas que converja hacia ese contorno. Generalmente guardamos este campo de fuerzas en una estructura de voxels. Un algoritmo de procesado que funciona muy bien es el algoritmo GradientVector Flow (GVF) de Chenyang Xu y Jerry L. Prince que se explica en el artículo [13].
3. Insertar en el campo de fuerzas un sistema de partículas organizado en superficie (generalmente se crea a partir de una malla esférica u ovalada) y aplicar las propiedades de la dinámica de partículas, para que éstas se dirijan hacia donde convergen las fuerzas, es decir hacia el contorno del objeto. Para suavizar el resultado generalmente los

sistemas de partículas utilizados crean una superficie elástica mediante resortes.

No explicaremos los algoritmos de los pasos 1 y 2. En futuras versiones de la librería OpenSIM con funcionalidades de procesado de la imagen, podrán verse más en detalle algoritmos de este tipo. A continuación ilustraremos estos pasos con unas figuras extraídas del artículo [13].

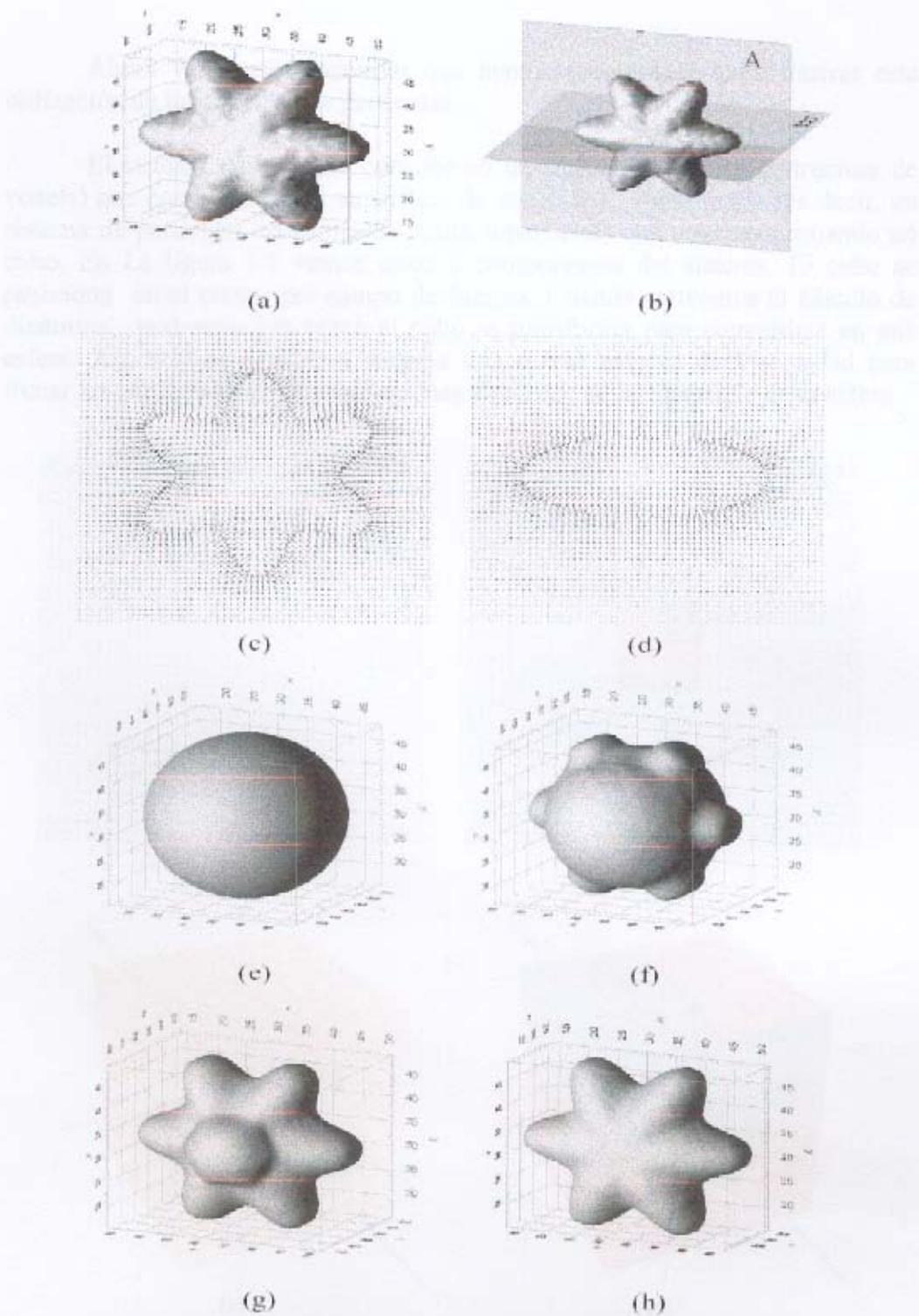


Figura 5.3.1 Reconstrucción de un Objeto 3D.

- (a) Objeto a reconstruir.
- (b) Plano en los cuales calcularemos el gradiente de fuerzas. Podrían ser las imágenes a partir de las cuales trabajariamos para reconstruir el objeto.
- (c) Campo de fuerzas (gradiente) generado con el método GVF sobre el plano A.
- (d) Campo de fuerzas (gradiente) generado con el método GVF sobre el plano B.
- (e) Malla (sistema de partículas) a partir del cual reconstruiremos el objeto.
- (f) Primeras iteraciones de la reconstrucción.
- (g) Objeto reconstruido a mitad.
- (h) Objeto reconstruido.

Ahora veremos el ejemplo que hemos programado para ilustrar esta utilización de los sistemas de partículas.

El sistema en cuestión consiste en un campo de fuerzas (estructura de voxels) que convergen en la superficie de una esfera, y una malla (es decir, un sistema de partículas en forma de malla superficial) con resortes formando un cubo. En La figura 5.2 vemos estos 2 componentes del sistema. El cubo se posiciona en el centro del campo de fuerzas. Cuando activemos el cálculo de dinámica, podremos ver cómo el cubo se transforma para convertirse en una esfera. Añadiremos a nuestro sistema una fuerza externa de viscosidad para frenar las partículas y evitar oscilaciones alrededor de la superficie de la esfera.

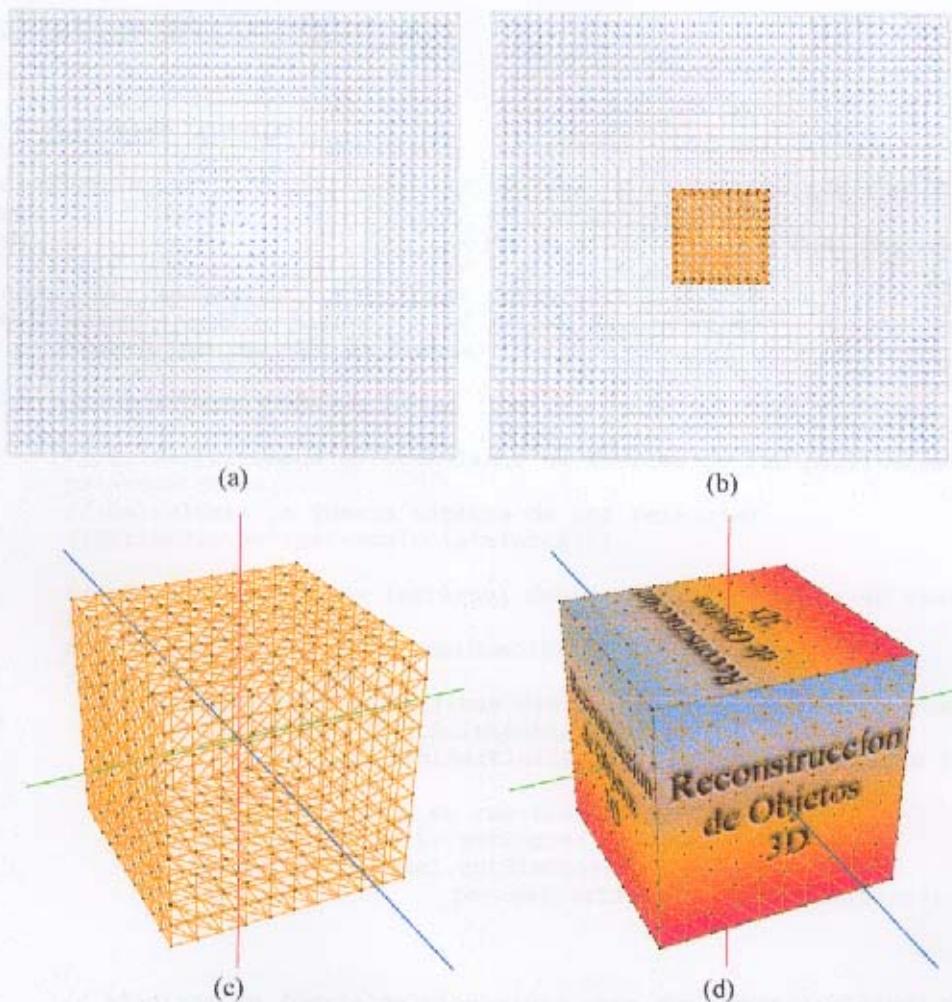


Figura 5.3.2 Ejemplo de reconstrucción 3D

- (a) Estructura de voxels, donde las fuerzas convergen hacia la superficie de una esfera.
- (b) Estructura de voxels mas el sistema de partículas.
- (c) Sistema de partículas con resortes.
- (d) Sistema de partículas más la malla texturizada a partir del la cual se ha creado.

Para conseguir el campo de fuerzas sin tener que extraerlo de algoritmos de procesado de imagen, hemos utilizado la función siguiente:

$$\mathbf{f}_{ijk} = \frac{\mathbf{C} - \mathbf{C}_{ijk}}{|\mathbf{C} - \mathbf{C}_{ijk}|} (|\mathbf{C} - \mathbf{C}_{ijk}| - r) k ,$$

donde

$\mathbf{f}_{ijk}$  es la fuerza que contiene el voxel i,j,k ,

$\mathbf{C}$  es el centro de la estructura de voxels,

$\mathbf{C}_{ijk}$  es el centro del voxel i,j,k

$r$  es el radio de la esfera que queremos reconstruir.

$k$  es una constante de control de la amplitud de la fuerza

### Programación con OpenSIM:

```
//-- variables Opensim
SpringSystem ss; // Sistema de partículas con resortes
Voxel voxel; // estructura de voxels
Mesh mesh; // Malla a partir de la cual creamos el
// sistema de partículas
Solver solver; // Integrador numérico
OpenGLOutput openGLOutput; // Salida de pantalla
// -- Función que calcula la fuerza

void calculateForce(ParticleSystem *ps, double t)
{
    // Reinicializamos el acumulador de fuerzas de las partículas
    ps->resetForce();
    // Calculamos la fuerza interna de los resortes
    ((SpringSystem*)ps)->calculateForce();

    // añadimos la fuerza (externa) debida a la estructura de voxels
    // en cada partícula
    for( int i = 0 ; i < ps->getNumParticles() ; i++)
    {
        // si la partícula se sitúa dentro de la estructura de voxels
        if( voxel.getSimObjectPointerAtPosition(
            ps->getParticleAt(i).getPosition()) != 0 )
        {
            // añadimos la fuerza en cuestión
            ps->getParticleAt(i).getForce() +=
                *((Vector3D_d*)voxel.getSimObjectPointerAtPosition(
                    ps->getParticleAt(i).getPosition()) );
        }
    }
    // añadimos la fuerza de viscosidad para que nuestro sistema no
    // oscile demasiado
    Force::viscosity(*ps, 0.2);
}
```

```

}

// -- En la función de inicialización del programa

// Importamos una malla en forma de cubo.
// El tamaño de una arista es de 100 unidades .
Import::ASEFileToMesh("./Meshes/cube.ase", mesh );

// Creamos una matriz de transformación, para escalar la malla a las
// dimensiones de nuestro sistema.
Matrix3D_d matrix;
matrix.set( 0.4,0,0,0,
           0,0.4,0,0,
           0,0,0.4,0,
           0,0,0,1);

// Aplicamos la transformación a la malla. Queda un cubo de lado 40
// unidades
mesh *= (matrix);

// Creamos el sistema de partículas con resortes a partir de la malla
// anterior.
ss.createFromMesh(mesh, 0, 0.002,0.001);
// Asignamos la malla al sistema de partículas para ver como se
// transforma la malla en una esfera.
mesh.assignToParticleSystem(ss);

// Inicializaremos la estructura de voxels
voxel.setOrigen(-100,-100,-100);
voxel.setSize(100,100,100);
voxel.setVoxelSize(2,2,2);

Point3D_d C;

// Centro de la estructura de voxels
C.set( voxel.getSize(SIM_X)*voxel.getVoxelSize(SIM_X)/2 +
        voxel.getOrigen().getValueAt(SIM_X),
        voxel.getSize(SIM_Y)*voxel.getVoxelSize(SIM_Y)/2 +
        voxel.getOrigen().getValueAt(SIM_Y),
        voxel.getSize(SIM_Z)*voxel.getVoxelSize(SIM_Z)/2 +
        voxel.getOrigen().getValueAt(SIM_Z));
// radio de la esfera a recuperar
float r = voxel.getSize(SIM_X)*voxel.getVoxelSize(SIM_X)/4;
// Calculamos la fuerza de cada voxel
for( int i = 0 ; i < voxel.getSize(SIM_X) ; i++)
{
    for(int j = 0 ; j < voxel.getSize(SIM_Y) ; j++)
    {
        for(int k = 0 ; k < voxel.getSize(SIM_Z) ; k++)
        {
            // Centro del voxel i,j,k
            Point3D_d Cijk( voxel.getVoxelSize(SIM_X) * ( i + 0.5 ) +
                            voxel.getOrigen().getValueAt(SIM_X),
                            voxel.getVoxelSize(SIM_Y) * ( j + 0.5 ) +
                            voxel.getOrigen().getValueAt(SIM_Y),
                            voxel.getVoxelSize(SIM_Z) * ( k + 0.5 ) +
                            voxel.getOrigen().getValueAt(SIM_Z)
                        );
            // Vector fuerza
            Vector3D_d v;

            // Creamos el vector a partir del puntero SimObject
            voxel.getSimObjectPointerAtIndex(i,j,k) = new Vector3D_d;

            // Aplicamos la formula de la fuerza
            v = (C - Cijk);
            // El 0.02 es un factor de control de la

```

```

    // amplitud de las fuerzas
    v = v.normalize() * (v.modul() - r)*0.02;

    // inserimos la fuerza en la estructura de voxels
    *((Vector3D_d*)voxel.getSimObjectPointerAtIndex(i,j,k)) = v;
}
}

// -- En el bucle principal del programa

// Calculamos las nuevas posiciones de las particulas
solver.rungeKutta4(ss, calculateForce );

// Dibujamos los elementos del sistema
openGLOutput.render(voxel);
openGLOutput.render(mesh, false, false, true, false, false, 0, true);
openGLOutput.render(ss, true, false, false, true );

```

Para los resultados de este ejemplo, ver la sección 6.

Figura 5.10. Evolución de la velocidad inicial de los partículas. Se observa que las velocidades iniciales de los partículas están dirigidas hacia el centro de la nube de partículas.

(a) Velocidad inicial de las partículas. Pueden más grande que las partículas y no tienen velocidad.

(b) La representación de las partículas que se mueven con una velocidad constante.

## 6. Resultados

Sistema eléctrico:

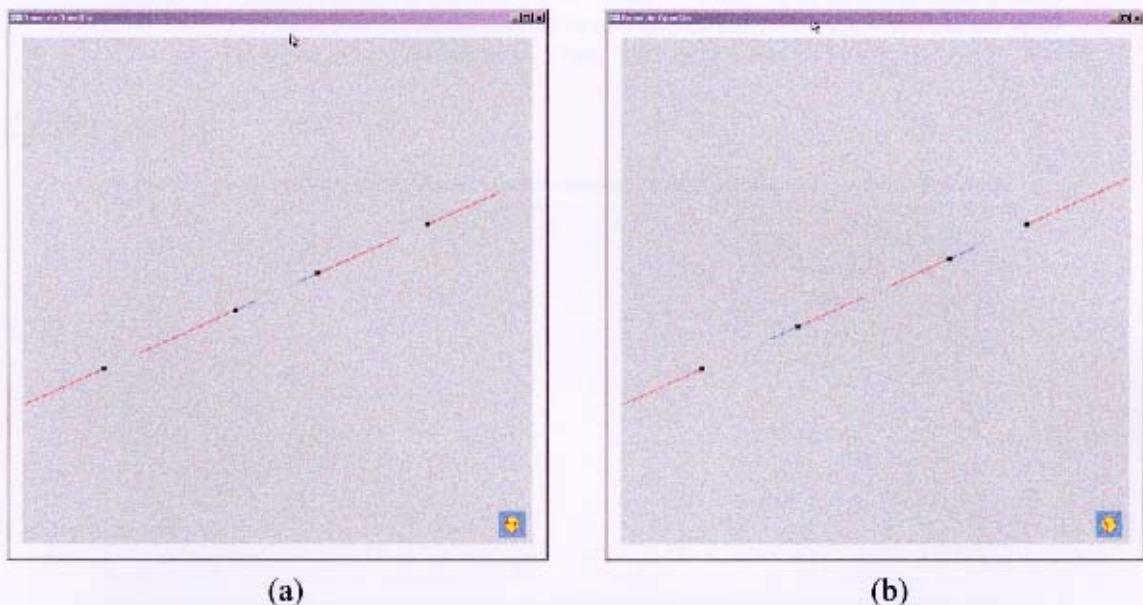


Figura 6.1 Simulación de partículas eléctricas. Las partículas de los extremos son fijas mientras que las centrales son libres. Todas son eléctricamente positivas. En rojo podemos ver el acumulador de fuerzas de las partículas y en azul sus velocidades.

- (a) La fuerza repulsión entre las partículas libres es mas grande que la de las partículas fijas sobre las libres.
- (b) La fuerza repulsión de las partículas fijas sobre las libres es mas grande que la de las partículas libres entre ellas.

### Sistema gravitacional:

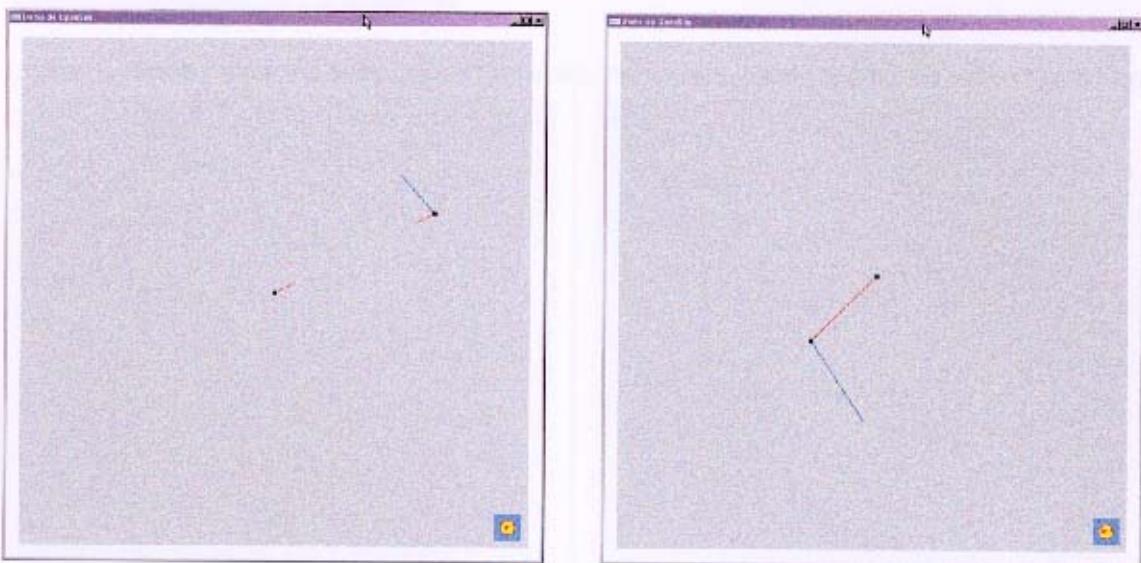


Figura 6.2 Sistema gravitacional de 2 partículas. La trayectoria de la partícula exterior es elíptica y la partícula central ocupa uno de sus focos. En rojo se observa la fuerza de las partículas y en azul sus velocidades.

### Cuerda elástica:

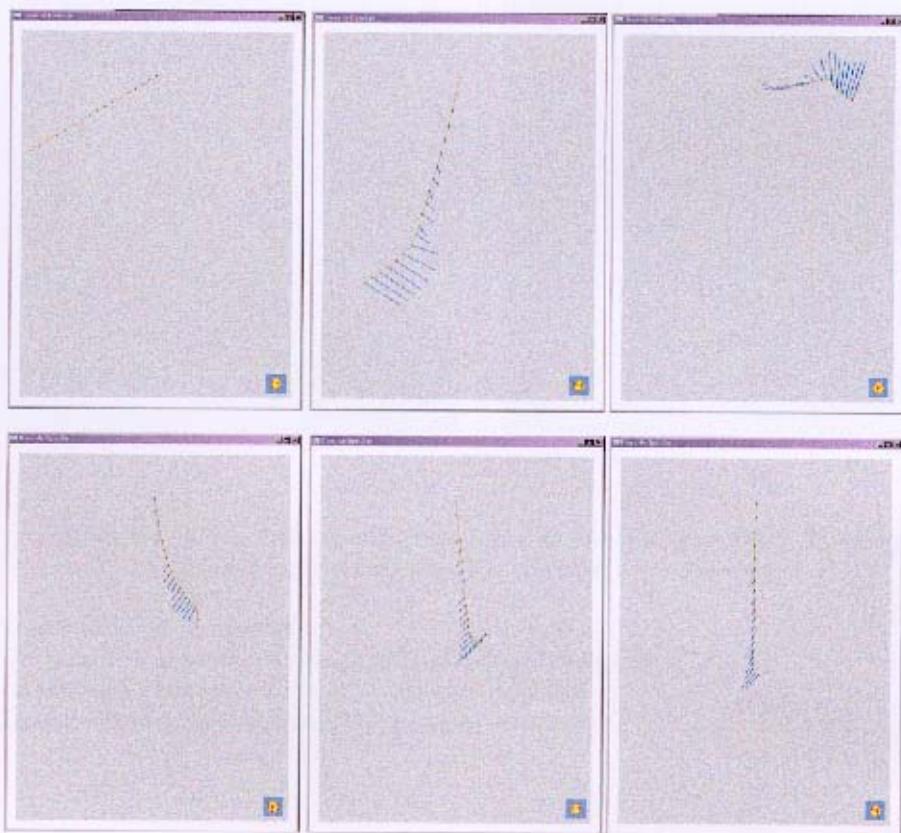


Figura 6.3 Simulación de una cuerda elástica modelada con 20 partículas y 19 resortes. La velocidad de las partículas se representa en azul.

### Colisiones con planos y triángulos:

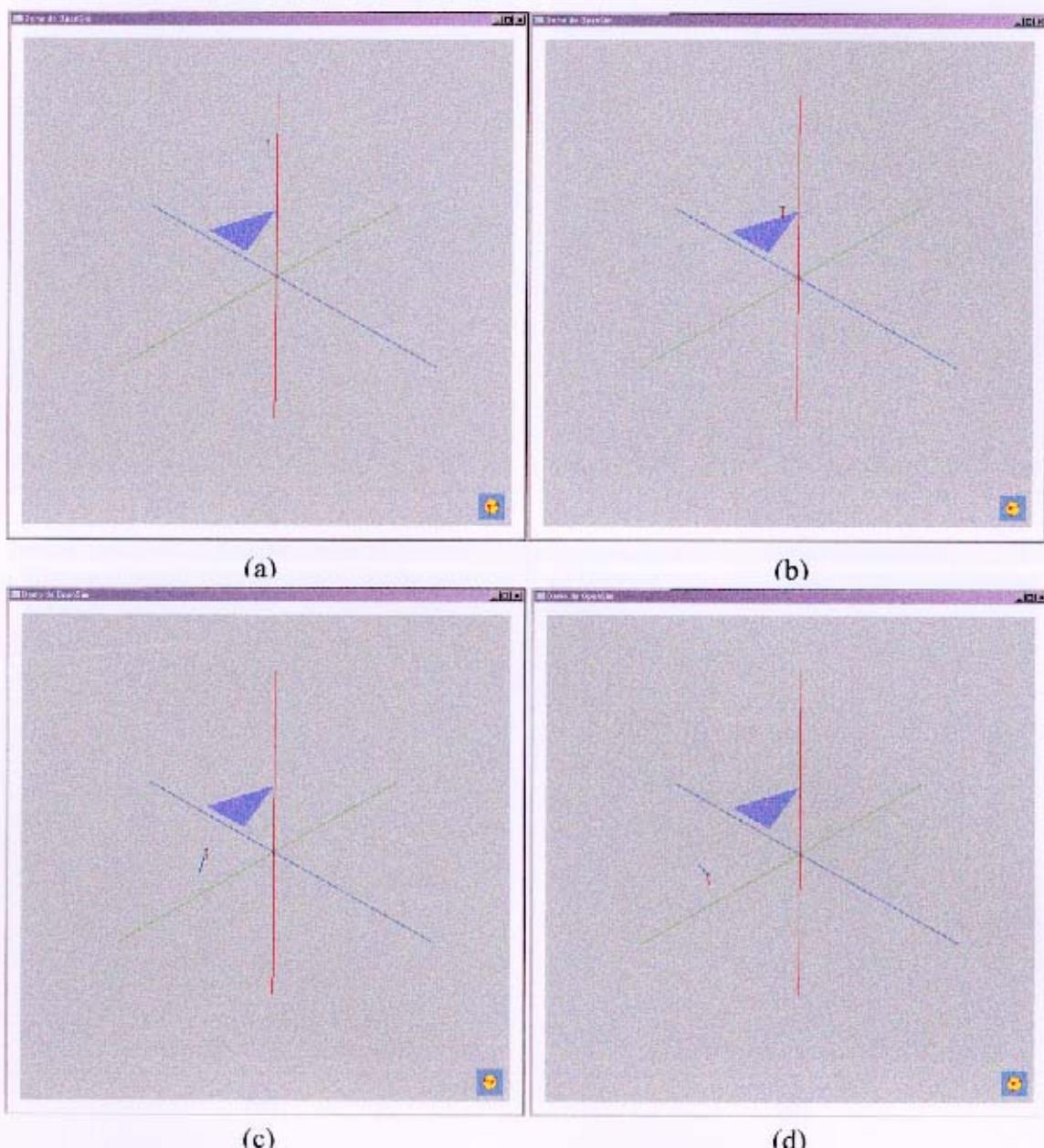


Figure 6.4 Tratamiento de colisiones sobre un triángulo y un plano. El triángulo está inclinado, y el plano tiene por ecuación  $y = 0$  (eje rojo).

- (a) La partícula cae en vertical.
- (b) Después de la primera colisión amortiguada con el triángulo.
- (c) La partícula a salido del triángulo y cae hacia el plano.
- (d) Después de la primera colisión amortiguada con el plano.

Agua:

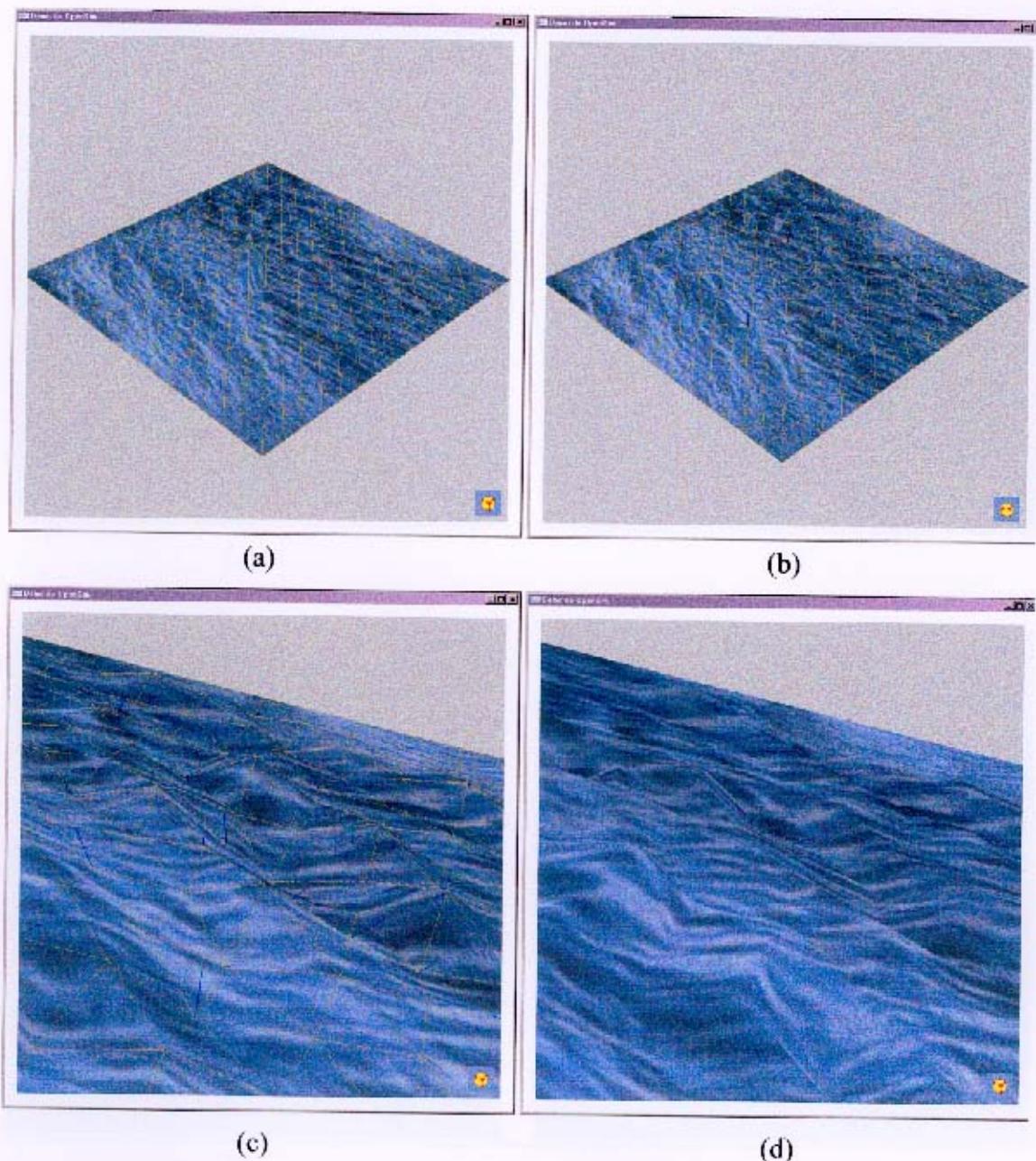


Figura 6.5 Simulación de la superficie del agua. En naranja se ve la estructura del sistema de partículas con resortes.

- (a) Estado inicial del sistema. Estiramos un vértice para perturbar la superficie.
- (b) La superficie se perturba creando ondulaciones.
- (c) Vista mas cercana de la superficie.
- (d) No representamos el sistema de partículas , solo la malla con textura.

Bola de fuego:

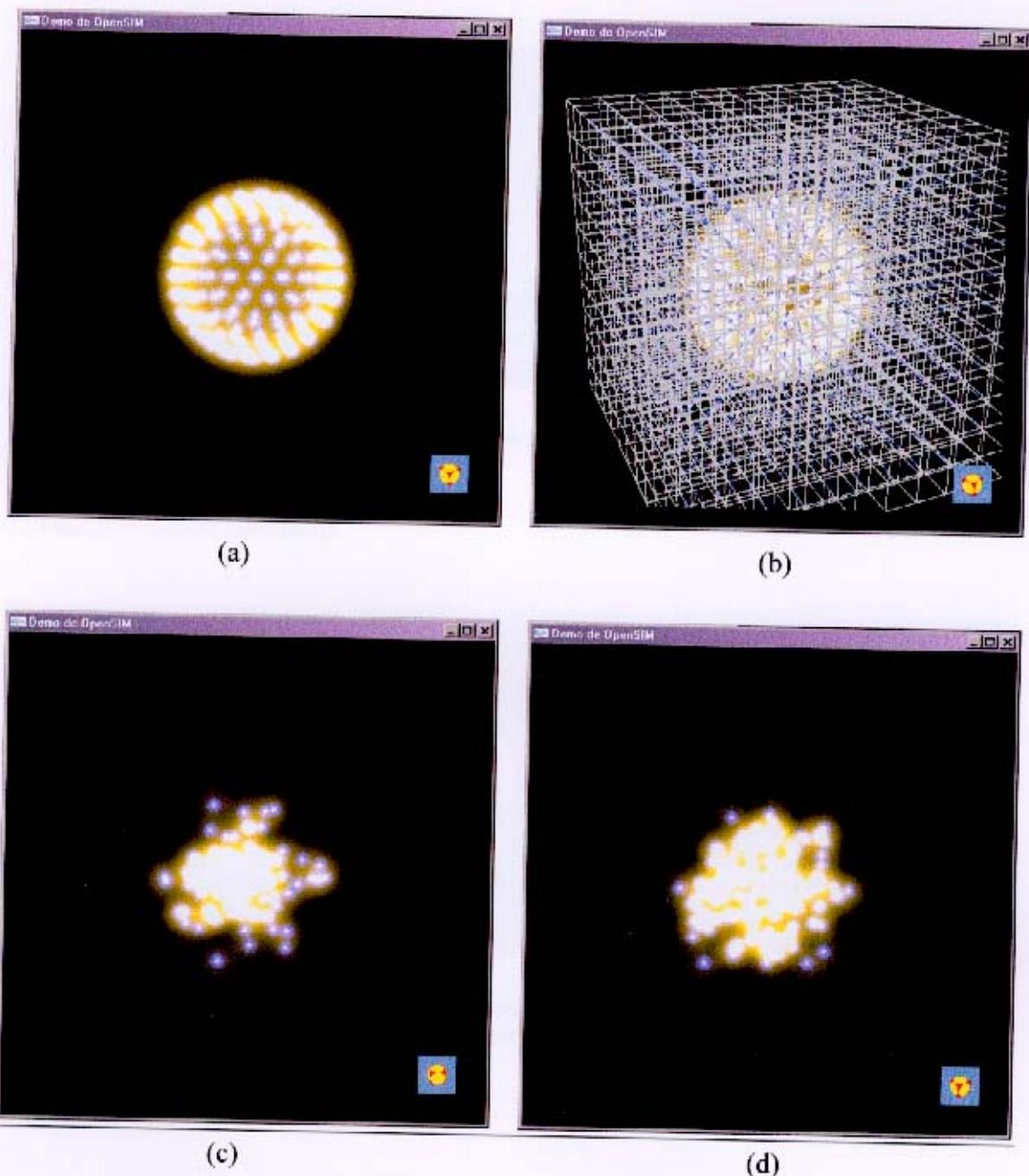


Figura 6.6 Simulación de una bola de fuego mediante una estructura de voxels de fuerzas concéntricas..

- Estado inicial del sistema de partículas.
- El sistema de partículas está dentro de la estructura de voxels.
- y (d) las partículas se quedan agrupadas formando una bola de fuego.

Explosión:

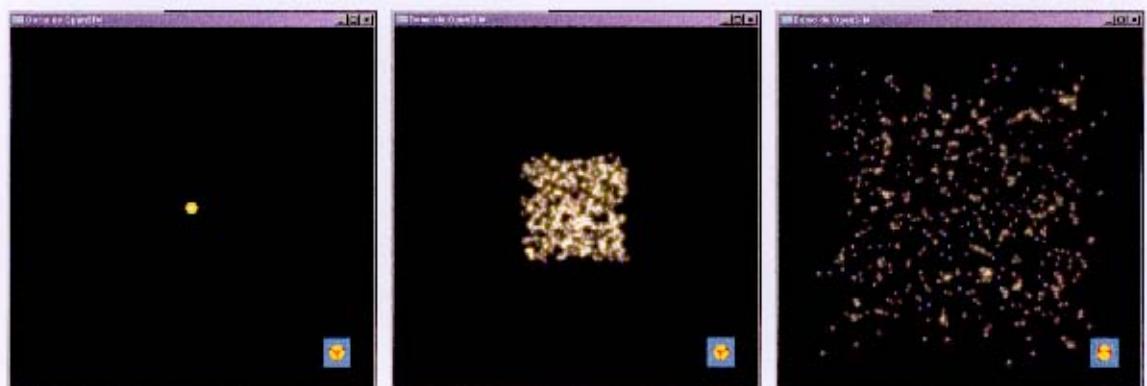


Figura 6.7 Simulación de una explosión.

Fuegos artificiales (bengala):



Figura 6.8 Simulación de un fuego continuo en dirección vertical. La fuerza de la gravedad hace caer a las partículas.

Simulación de ropa:

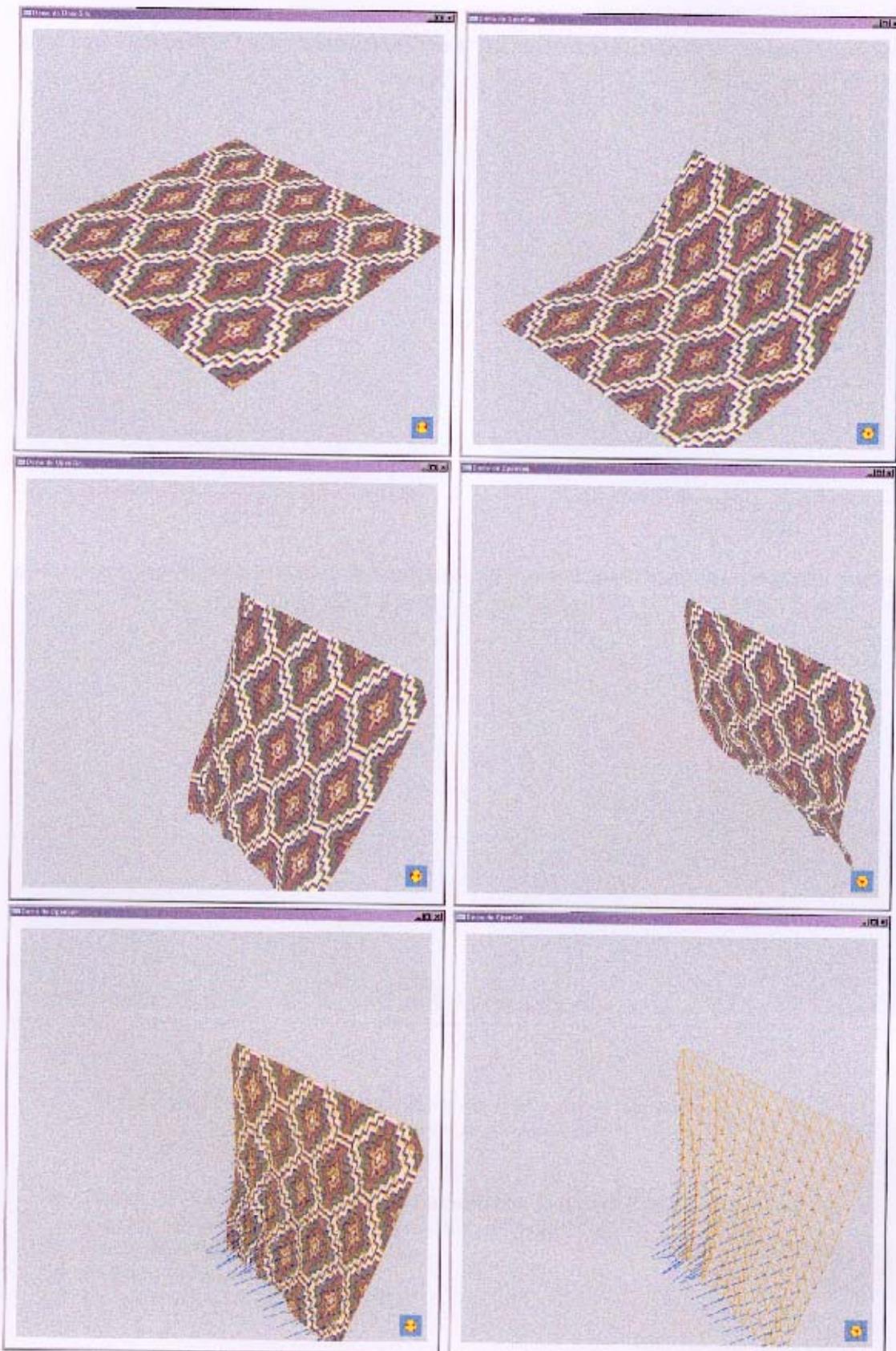


Figura 6.9 Simulación de una tela con una malla de partículas y resortes.

### Reconstrucción 3D de una esfera:

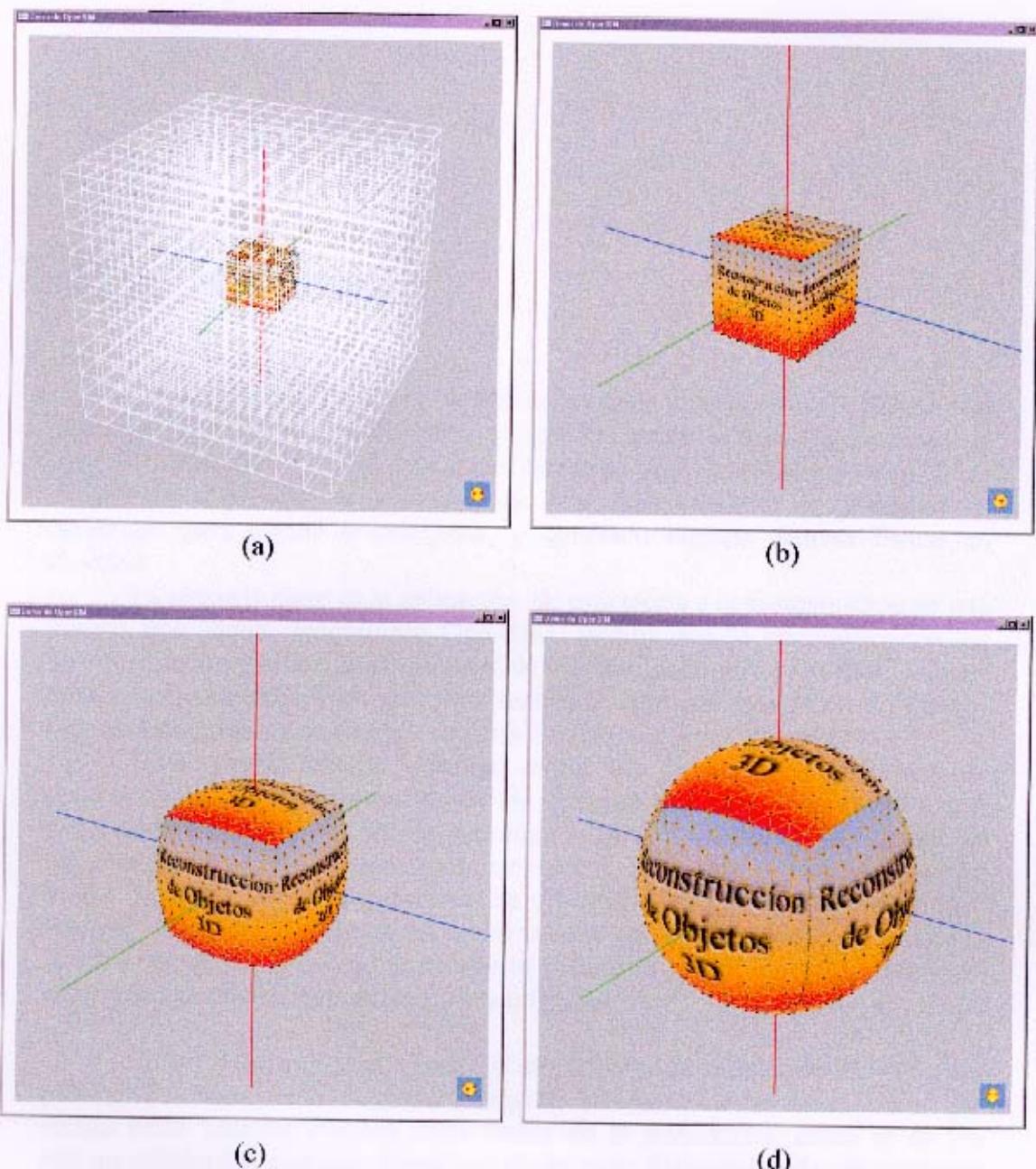


Figura 6.10 Reconstrucción de una esfera a partir de una malla de partículas con resortes en forma de cubo.

- Sistema de partículas situado dentro de la estructura de voxels. Para la reconstrucción de ha utilizado una estructura mucho mas detallada ( $100 \times 100 \times 100$ ).
- Estado inicial del sistema de partículas.
- Primeras iteraciones del sistema.
- Reconstrucción del Objeto, una esfera.

## 7. Conclusiones

La primera parte del proyecto, donde se explica la teoría computacional de partículas, se ha presentado como un posible curso sobre la simulación de objetos deformables; se han explicado, demostrado, e ilustrado los métodos de integración utilizados en simulaciones dinámicas, resuelto el problema de colisiones para planos y triángulos, y detallado algunas fuerzas físicas en concreto.

La segunda parte es la aplicación de esta teoría a la programación de una librería que hemos denominado OpenSIM. Hemos podido comprobar con los ejemplos de la sección 4 la simplicidad de uso de esta librería y sus posibilidades para la simulación de objetos deformables. Todos los apartados de la teoría (menos las funciones de energía, sección 3.5) han quedado ilustrados.

Como hemos podido verificar, existe una diferencia notable entre el sistema físico y su representación. En los ejemplos de fuego, el resultado visual viene dado por la aplicación de texturas de manera adecuada. La librería no resuelve estos problemas, sino que deja al usuario la libertad de la representación de las partículas según el tipo de salida que utilice (OpenGL o DirectX). Esta decisión de diseño permite no limitar al usuario en su programación y le deja la libertad de complicar o mejorar el aspecto gráfico de su aplicación en función de sus conocimientos de la librería gráfica utilizada.

En su primera versión, OpenSIM es un motor de simulación de partículas pero su ambición es la de poder competir con librerías profesionales en las cuales están tratados muchos otros temas de la simulación, como el de los objetos sólidos por ejemplo. Pensamos añadir estas funcionalidades en versiones futuras. Como se explicó en un principio, OpenSIM intentará también ser una base para múltiples proyectos de simulación, lo cual permitirá que la librería se vaya completando.

## 8. Líneas futuras

OpenSIM está sólo en su primera versión. La próxima etapa es la de entender la teoría necesaria para la simulación de objetos sólidos e incorporarla a la librería. De esta forma en su próxima versión, OpenSIM se convertirá en un motor de simulación gráfica completo, es decir, que tratará tanto los objetos deformables como los rígidos.

Sin embargo la librería tiene también otra función: la de unificar otros campos de la investigación, como el procesado de la imagen o la inteligencia artificial en un mismo entorno de programación. Estos temas pueden ser muy útiles conjuntamente utilizados con la simulación 3D. Por ejemplo, imaginemos que se quiere programar un robot con inteligencia para moverse en un cierto entorno; simularíamos las acciones del robot de forma realista con el motor de simulación mientras que con algoritmos de procesado de la imagen se podría recuperar la información pertinente del entorno y aplicarla a nuestros métodos de inteligencia artificial. También mediante estos temas de inteligencia artificial y simulación se pueden comprender fenómenos de la vida real. Los trabajos de Karl Sims [21] en esta dirección ilustran perfectamente como ciertos comportamientos reales provienen de una optimización de los movimientos respecto al entorno.

En un futuro esperamos conseguir resultados similares con la librería OpenSIM.

## 9. Bibliografía

### Referencias de Internet:

- [1] 3D StudioMax, programa de animación 3D,  
url: <http://www.discreet.com/products/3dsmax/>.
- [2] Macromedia Director, programa para la creación de productos multimedia,  
url: <http://www.macromedia.com/software/director/>.
- [3] Karma de Mathengine, motor de simulación dinámica,  
url: <http://www.mathengine.com/>.
- [4] OpenGL, librería gráfica de programación,  
url: <http://www.opengl.org/>.
- [5] Glut library, conjunto de funciones para la gestión de ventanas con OpenGL y otras utilidades.  
url: <http://www.opengl.org/developers/documentation/glut.html>.
- [6] DirectX, librería de programación de dispositivos de Microsoft.  
url: [www.microsoft.com/directx/default.asp](http://www.microsoft.com/directx/default.asp).
- [7] Nehe, pagina de referencia para la programación de gráficos 3D con OpenGL,  
url: <http://nehe.gamedev.net/>.
- [8] C++, lenguaje de programación orientado a objetos,  
url: <http://www.cuj.com/>.
- [9] Java, lenguaje de programación orientado a objetos,  
url: <http://java.sun.com/>.
- [10] HTML, lenguaje para la creación de paginas Web.  
url: <http://www.w3.org/MarkUp/>.
- [11] Object Outline, programa que genera una documentación automática a partir de los comentarios en el código fuente (C o C++) de nuestra aplicación.  
url: <http://www.bbeesoft.com/>

### **Referencias de artículos:**

- [12] David Baraff y Andrew Witkin. Particle System Dynamics, del curso Physically Based Modeling: Principles and Practice presentado en el SIGGRAPH 97.  
url: <http://www-2.cs.cmu.edu/~baraff/sigcourse/index.html>
- [13] Chenyang Xu y Jerry L.Prince. Snakes, Shapes, and Gradient Vector Flow. *IEEE Transactions on Image Processing*, 359 – 369, marzo 1998  
url: <http://iacl.ece.jhu.edu/projects/gvf/>

### **Referencias de libros:**

- [14] Bjarne Stroustrup. El lenguaje de programación C++. Addison-Wesley, segunda edición, 1993.
- [15] OpenGL Programming Guide. Addison-Wesley, tercera edición, 1999.
- [16] OpenGL Reference Manual. Addison-Wesley, tercera edición, 1999.
- [17] Bradley Bargen, Terence Peter Donnelly, Inside Directx (Microsoft Programming Series).

### **Otras referencias de interés:**

- [18] Numerical Recipes, libro online de métodos numéricos.  
url: <http://www.nr.com/>
- [19] Programs from *Numerical Methods for Physics* (Second Edition).  
url: <http://www.algarcia.org/nummeth/Programs2E.html>
- [20] Paul Bourk Home Page.  
url: <http://astronomy.swin.edu.au/pbourke/>
- [21] Karl Sims Home Page.  
<http://www.genarts.com/karl/>