



UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES ACATLÁN

“MODELADO GRÁFICO DE UN CUERPO
NEUMÁTICO CON OPENGL A BASE DE
ECUACIONES DIFERENCIALES”

T E S I S

QUE PARA OBTENER EL GRADO DE:

LICENCIADO EN MATEMÁTICAS
APLICADAS Y COMPUTACIÓN

P R E S E N T A

JORGE ANTONIO GARCÍA GALICIA

DIRECTOR DE TESIS:

DR. MARÍA DEL CARMEN VILLAR PATIÑO

México, D.F.

2008

*Para Onatta
por estar siempre en mi corazón.*

Agradecimientos

Agradezco a la Facultad de Estudios Superiores de Acatlán, por ser mi casa durante tantos años y por darme la formación profesional y humana que me acompañará durante toda mi vida.

Agradezco también a la jefatura del programa de Matemáticas Aplicadas y Computación por estar siempre dispuestos a ayudarme y por responder mis dudas de la manera más atenta posible.

Agradezco al Macroproyecto de Tecnologías de Información y la Facultad de Ciencias de la UNAM. En particular, a la UDeSyTI por permitirme utilizar sus instalaciones y por todos los recursos materiales con los que me apoyaron.

A mis sinodales de tesis: Carmen Villar, Francisco Patiño, Jaime Vergara, Victor Palencia y Teresa Carrillo; por todos sus comentarios y correcciones que hicieron de éste un mejor trabajo.

A mi directora de tesis: Carmen Villar; por haberme apoyado mucho más de lo que era su deber durante toda la duración de este trabajo.

Y por supuesto, gracias...

A mis compañeros de MAC, del TKD y de la FES Acatlán en general; por todas las pláticas y momentos compartidos.

A todos mis amigos y familiares, por siempre estar al pendiente de mí.

A mi familia, por apoyarme en todos los retos y proyectos que he emprendido durante toda mi vida.

A Hipatia, por estar a mi lado durante esta reedición de tesis.

A Onatta, por su amor, apoyo y por ser mi alma gemela y compañera de vida. Me complementas y me haces ser una mejor persona siempre.

Resumen

Este trabajo trata de cómo crear una simulación gráfica de un cuerpo flexible. El cuerpo flexible, está construido –como es costumbre en graficación por computadora– por una malla tridimensional. Además, en éste trabajo se asume que las aristas de la malla son resortes y que el cuerpo flexible está lleno de aire.

Para hacer una animación basada en Física, se definen un conjunto de fuerzas que actúan sobre los vértices de la malla. Después, se usa la fuerza acumulada para integrar la ecuación de Newton y obtener las velocidades y posiciones de los vértices en la escena.

Las fuerzas aplicadas son: la gravedad, los resortes-amortiguadores y la fuerza de presión. Esta última, es resultado de usar la ecuación del gas ideal.

Se describe a detalle, la creación de un programa que implementa todas estas ideas para hacer una animación interactiva. La implementación se hizo en lenguaje C++ y se usa OpenGL como biblioteca para desplegar gráficos.

Se hacen un conjunto de pruebas para comprobar el correcto comportamiento cualitativo en la implementación. Y por último, se hace un análisis del desempeño del programa; donde se estima que secciones son las computacionalmente más costosas.

Modelado Gráfico de un Cuerpo Neumático con OpenGL a Base de Ecuaciones Diferenciales © 2008 por Jorge Antonio García Galicia está licenciado bajo Attribution 4.0 International.

Para ver una copia de esta licencia, visite creativecommons.org/licenses/by/4.0/

Índice general

Introducción	1
1 Sistemas físicos y modelación matemática	5
1.1 Un cuerpo en caída libre	6
1.2 El sistema masa-resorte	10
1.2.1 Oscilador armónico simple	11
1.2.2 Oscilador armónico amortiguado	14
1.2.3 Masa-resorte en más de una dimensión	17
1.2.3.1 La fuerza del resorte	19
1.2.3.2 La fuerza del amortiguador	20
1.3 El modelo del gas ideal	22
1.3.1 La termodinámica	23
1.3.2 Un modelo simple	23
1.3.3 La presión de un gas	24
1.3.4 Un ejemplo ilustrativo	26
2 Construcción del algoritmo de simulación	33
2.1 Diseño del experimento	34
2.2 Simulaciones gráficas basadas en física	34
2.2.1 Esbozo general del algoritmo	35
2.2.1.1 Inicializar variables	36
2.2.1.2 Dibujar escena	37
2.2.1.3 Acumular fuerzas	37
2.2.1.4 Integrar la ecuación de Newton	38
2.2.1.5 Detectar y resolver colisiones	38
2.2.1.6 Interacción del usuario	39
2.3 El sistema de fuerzas	39
2.4 Área, volumen y vectores normales	41
2.4.1 Cálculo de áreas	41
2.4.2 Cálculo de volúmenes	42
2.4.2.1 Volúmenes aproximados	43
2.4.2.2 Volúmenes Exactos	43

2.4.3	Vectores normales	44
2.5	Integrar la ecuación de Newton	45
2.5.1	El método de Euler	47
2.5.1.1	El error del método de Euler	49
2.5.1.2	Ventajas y desventajas del método de Euler	49
2.5.2	El método de Runge Kutta	50
2.5.2.1	El error del método de Runge Kutta	52
2.5.2.2	Ventajas y desventajas del método de Runge Kutta . .	53
2.6	Cómo enfrentar la colisión de los cuerpos	53
2.6.1	La detección de las colisiones	54
2.6.1.1	Un bounding volume	54
2.6.1.2	Uniendo diferentes geometrías	55
2.6.2	La respuesta de las colisiones	56
2.6.2.1	Separar un vector en componentes normal y tangencial	56
2.6.2.2	Colisiones elásticas	57
2.6.2.3	Colisiones inelásticas	61
3	Implementación del modelo	63
3.1	Diseño de clases	64
3.2	Creación del cuerpo flexible	67
3.3	La física del modelo	70
3.3.1	La fuerza de gravedad	71
3.3.2	La fuerza de los resortes	71
3.3.3	La fuerza del gas	73
3.4	Los métodos numéricos	76
3.4.1	El método de Euler	76
3.4.2	El método de Runge-Kutta	76
3.5	El manejo de las colisiones	78
3.5.1	La rutina de las colisiones	78
3.5.2	La detección de la colisión	79
3.5.3	La respuesta de la colisión	80
4	Diseño del experimento	81
4.1	Definición del sistema	81
4.1.1	Características del modelo	81
4.1.1.1	Constantes del experimento	82
4.1.1.2	Variables físicas del experimento	83
4.1.1.3	Opciones de control y visualización	86
4.1.2	Características del entorno de pruebas	87

4.1.2.1	Software	87
4.1.2.2	Hardware	87
4.2	Características físicas del modelo	88
4.2.1	Probando la gravedad	88
4.2.2	Probando la fuerza del gas	91
4.2.3	Probando los resortes amortiguadores	93
4.3	Ánalisis del desempeño	95
4.3.1	Desempeño del programa	96
Conclusiones		101
Apéndice		103
Bibliografía		107

Índice de figuras

1.1	Ejemplo de caída libre	7
1.2	Diagrama de cuerpo libre	8
1.3	Masa unida por un resorte	12
1.4	Ley de Hooke para resortes	13
1.5	Plano fase del oscilador armónico amortiguado	17
1.6	Masas libres en el espacio unidas por un resorte	18
1.7	Ejemplo de vectores de posición	19
1.8	Cuadrilátero	26
1.9	Cuadrilátero localizado en el espacio	27
1.10	Cálculo del área de un cuadrilátero	28
1.11	Vectores de presión incorrectos	31
1.12	Vectores de presión correctos	32
2.1	Diagrama del modelo masa, resorte, y presión	35
2.2	Diagrama de flujo de la simulación	36
2.3	Colisión elástica	58
2.4	Separar componente tangencial y normal de un vector	59
3.1	Diagrama de clases	66
4.1	Ejemplo del programa en ejecución	82
4.2	Menú de usuario del programa	84
4.3	Experimento: Fuerza de gravedad	89
4.4	Experimento: $g > 0$	90
4.5	Experimento: Apagar la fuerza de gravedad	91
4.6	Experimento: Fuerza del gas - cuerpo rígido	92
4.7	Experimento: Variar la fuerza del gas	93
4.8	Experimento: Quitar la fuerza del amortiguador	94
4.9	Explosión por inestabilidad numérica	94
4.10	Experimento: Fuerza del resorte	95

Índice de cuadros

1.1	Ejemplo sobre cómo calcular la fuerza de presión	31
2.1	Resumen de las fuerzas que actúan sobre cada partícula	40
2.2	Evolución histórica de los métodos numéricos	46
2.3	Condiciones de la respuesta a las colisiones	57
4.1	Valores de las constantes durante el experimento	83
4.2	Valores por defecto de las variables físicas	85
4.3	Resumen de medidas por frame	98
4.4	Resumen de medidas de evaluación de \mathbf{F}	98
4.5	Proporción de cálculos por frame	99
4.6	Proporción de cálculos de \mathbf{F}	99

Introducción

Uno de los retos más constantes en las ciencias de la computación, y en especial del área de graficación por computadora, ha sido el de alcanzar cada vez mayor realismo. Sin embargo, este objetivo comúnmente se opone al de hacer modelos simples que requieran de poco poder de cómputo.

Es cierto que cada vez tenemos computadoras más poderosas, pero como seres humanos que somos siempre hemos querido muchas cosas más de las que tenemos. Hoy en día el que una simulación se pueda ejecutar rápidamente en una computadora personal es casi tan importante como el que esa simulación se vea real.

Modelos que se sabe son bastante efectivos, como las ecuaciones de Navier-Stokes, también son computacionalmente muy costosos. Por lo tanto, no pueden ser ocupados en simulaciones que requieren de alcanzar tiempo real.

Los videojuegos son un claro ejemplo de simulaciones que requieren de una adecuada combinación de realismo y rapidez de ejecución. Dentro de esta área, se dice que se alcanza tiempo real cuando el programa responde más rápido de lo que el usuario puede darle instrucciones. El objetivo principal de este trabajo es encontrar un modelo que conjunte el mayor realismo posible, sin olvidar alcanzar *tiempo real*.

Hace tiempo, cuando buscaba algún tema interesante para realizar mi trabajo de titulación, no tenía una idea clara de lo que quería hacer. Por un lado sabía qué me gustaba de las materias de mi licenciatura, también sabía que quería hacer una tesis que de alguna manera tuviera que ver con el área de gráficas por computadora.

Afortunadamente para mí, eso fue lo único que necesité para que la maestra Carmen Villar se interesara en asesorar mi trabajo de titulación, aun cuando no tenía un tema bien definido. Después de buscar en diversas fuentes de información llegamos al sitio

web de Maciej Matyka, sobre simulación basada en física, en donde pude consultar el artículo [11]. Desde el primer momento este artículo llamó mi atención, el modelo propuesto gozaba de las tres características más deseables en un modelo de simulación: era sencillo de entender, sencillo de implementar y era visualmente muy agradable.

Teníamos entonces una idea de donde comenzar, sólo nos hacía falta un problema que valiera la pena modelar. Discutiendo en el laboratorio de Linux de la FES Acatlán, quizás un poco inspirados en la película “American Beauty”, tuvimos la idea del sistema que dio origen al presente trabajo. Al final creo que el resultado poco tiene que ver con la inmortal escena, pero estoy convencido de que al menos para mí ha sido un camino igual de inspirador.

Otra meta que quería alcanzar era utilizar Software Libre en todo este trabajo. Sin duda, el utilizar una biblioteca como OpenGL me favoreció para poder alcanzar esta meta, al tratarse de una especificación libre y multiplataforma. La mayor parte del trabajo fue realizada bajo un sistema GNU/Linux y el texto fue escrito en L^AT_EX. Considero que este objetivo fue cumplido.

A lo largo del desarrollo, tanto del programa como del trabajo escrito, me encontré con muchísimos obstáculos. Algunos se debieron a mi desconocimiento de ciertas áreas de la física, y otros simplemente a mi falta de habilidades de programación. Sin embargo, estoy convencido que en ambos aspectos este trabajo me ayudó a superarme.

La pregunta más importante que ahora trato de responderme es: ¿a quién le sirve este trabajo? Indudablemente me sirvió a mí el escribirlo, y quiero pensar que también le servirá a cualquier persona que lo lea. ¿Quién puede ser éste lector potencial? Es un trabajo de animación por computadora, por lo que le debe servir a cualquier persona que desee hacer una animación de cuerpos flexibles. También creo que es un trabajo que modela adecuadamente las leyes físicas del fenómeno, así que se puede beneficiar de él cualquier persona que desee aprender o enseñar cómo funcionan los cuerpos neuromáticos. Por último, le sirve a cualquier estudiante que busque alguna aplicación de las matemáticas y la física en la animación.

La manera como decidí organizar este trabajo es la siguiente:

En la primera parte se revisó el marco teórico. Éste trata principalmente de aquellas partes de la literatura, que aunque ya están escritas, quizás no sean tan obvias para estudiantes de la licenciatura de Matemáticas Aplicadas y Computación. Este capítulo

fue por mucho el más divertido de escribir.

En el siguiente capítulo, ya adentrado en la situación a modelar, traté en mayor detalle las estrategias de implementación del modelo. Es un capítulo en el que intenté plasmar, sobre todo, las cosas que me fueron más difíciles en el desarrollo de la investigación y cómo fue que encontré soluciones a ellas.

El tercer capítulo trata de cómo se hizo el programa en C++ para esta simulación. Aunque el código que aquí escribo puede ser mejorado de muchas maneras, es una solución que funciona. Y si bien no me preocupé en optimizar el código, si se ganó un poco de claridad. Además, considero dado el propósito de este trabajo, que la programación no es *tan* importante.

En el capítulo final, presento las pruebas y juegos a los que sometí mi programa una vez que terminé el desarrollo. Este capítulo fue por mucho el más gratificante de escribir. Al escribir este capítulo aprendí dos cosas: que siempre hay más cosas que hacer después de que uno piensa que ha terminado algo y que los resultados no siempre son los que uno espera. De hecho, el desarrollo de esta investigación, de la que fui parte desde su nacimiento, me hizo llevarme unas cuantas sorpresas en su edad madura.

Por último, sólo me queda esperar que los lectores de este trabajo encuentren al leerlo, un poco de la diversión que me dejó a mí escribirlo.

Capítulo 1

Sistemas físicos y modelación matemática

“In fact, computer graphics is arguably the best example of a practical area where so much mathematics combines so elegantly.”

Samuel R. Buss

*3-D Computer Graphics A Mathematical Introduction
with OpenGL*

Uno de los principales fenómenos naturales que más se ha estudiado a lo largo de la historia es el del movimiento. La rama de la física que se encarga de estudiar este fenómeno es la *mecánica*. Hay muchos enfoques para estudiar el movimiento. La mecánica puede dividirse a su vez en: *estática*, *cinemática* y en *dinámica*. La estática estudia la ausencia del movimiento. La cinética se encarga de estudiar la manera de describir el movimiento. La dinámica estudia las causas del movimiento, es decir estudia las fuerzas que producen el movimiento.

El movimiento que estamos tratando de modelar puede pensarse como una combinación de movimientos más simples, para los cuales ya existen modelos matemáticos. Por lo que en este capítulo presentaré estos movimientos por separado, con el fin de entender las causas que los producen. Después en el capítulo siguiente uniré estos sistemas simples en uno más complejo, que es el que a fin de cuentas nos interesa modelar.

El primer movimiento que presentaré es el de un cuerpo que se deja caer en el vacío sin imprimirle ninguna fuerza, y que por lo tanto, sólo actúa sobre él la fuerza de gravedad. Este movimiento ha sido muy estudiado y es conocido como *caída libre*.

El siguiente movimiento, es también muy conocido, se conoce como *ley de Hooke*. Éste es el movimiento de algunas masas unidas por un resorte y se modela con una ecuación diferencial.

Por último, analizaré el fenómeno de la presión del gas, con el modelo más simple de todos, que es conocido como la ecuación de estado de la *ley de los gases ideales*. En el cual, se supone un gas hipotético en el que sus moléculas no interactúan entre sí.

En este capítulo, trataré estos sistemas por separado para poder comprenderlos cualitativamente. Así será más clara la manera en que los combinaré en el siguiente capítulo para formar un modelo. Y finalmente, con este modelo trabajar la situación que deseo analizar.

1.1 Un cuerpo en caída libre

Imagínese una piedra que descansa cerca de la orilla del quicio de una ventana. La piedra es empujada horizontalmente hasta que resbala de la orilla y se cae, (ver Figura 1.1). Éste es un ejemplo de caída libre.

Más formalmente, la caída libre consiste en modelar un cuerpo sólido, con masa uniformemente distribuida y despreciando la fricción del aire. Es decir, la única fuerza que actúa sobre este cuerpo es la *fuerza de gravedad*.

En la antigüedad, Aristóteles pensaba que este movimiento era directamente proporcional a la masa. Él afirmaba que dada una misma altura, un cuerpo con el doble de masa caería el doble de rápido. Esta idea fue aceptada durante varios siglos. No fue hasta la llegada del método científico que se empezó a cambiar este punto de vista.

Se cuenta una anécdota con ciertos toques de leyenda. Se dice que Galileo dejó caer en un mismo instante una pelota de madera de un kilo de peso y una bala de cañón de diez kilos de peso desde la torre inclinada de Pisa, y la gente pudo apreciar como estos cayeron casi al mismo tiempo. Con el simple hecho de un experimento la teoría aristotélica quedaba refutada. Al parecer antes de los pensadores y filósofos del renacimiento, nadie

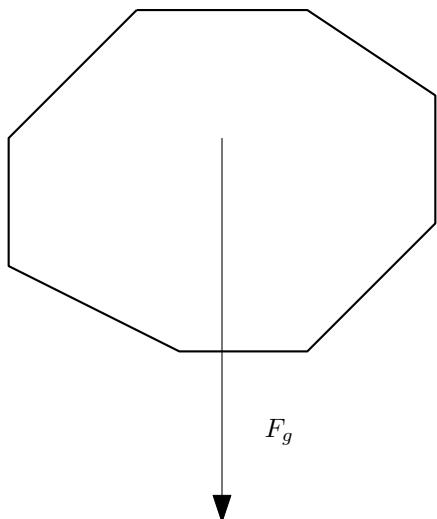


Figura 1.1: Una piedra en caída libre, la única fuerza que actúa sobre ella es la de la gravedad.

se había preguntado si los conocimientos que prevalecían eran verdaderos; es decir, nadie había hecho un experimento para probar si algo era de verdad como se decía. La simple autoridad de Aristóteles bastaba para que se diese por hecho.

Fue después de Galileo en 1685 que Newton explicó con más detalle este movimiento, con su *ley de la gravitación universal*. Newton fundamentó la mecánica con sus leyes del movimiento y la ley de la gravitación universal. En las primeras, se dice que un objeto permanece en su estado de reposo o de movimiento rectilíneo uniforme a menos que una fuerza lo perturbe. Esta fuerza perturbadora le produce al objeto un cambio de velocidad, es decir, una aceleración, en mecánica de Newton las fuerzas producen *aceleración*. La ley de la gravitación universal dice que todos los cuerpos se atraen. También dice que la fuerza de atracción entre dos cuerpos es directamente proporcional al producto de sus masas e inversamente proporcional al cuadrado de sus distancias.

A partir de esta idea, si la fuerza cambia con el tiempo, también lo hará la aceleración. Sin embargo por suerte para nosotros, el movimiento de caída libre imprime siempre *aproximadamente* la misma aceleración, es decir, la fuerza de gravedad actúa sobre todas las cosas con *casi* la misma fuerza. Una consecuencia de la ley de la gravitación universal y de que la masa de la Tierra sea muy grande en comparación de las cosas que estamos acostumbrados ver caer.

Gracias a que la fuerza que atrae las cosas a la tierra es casi constante, el problema

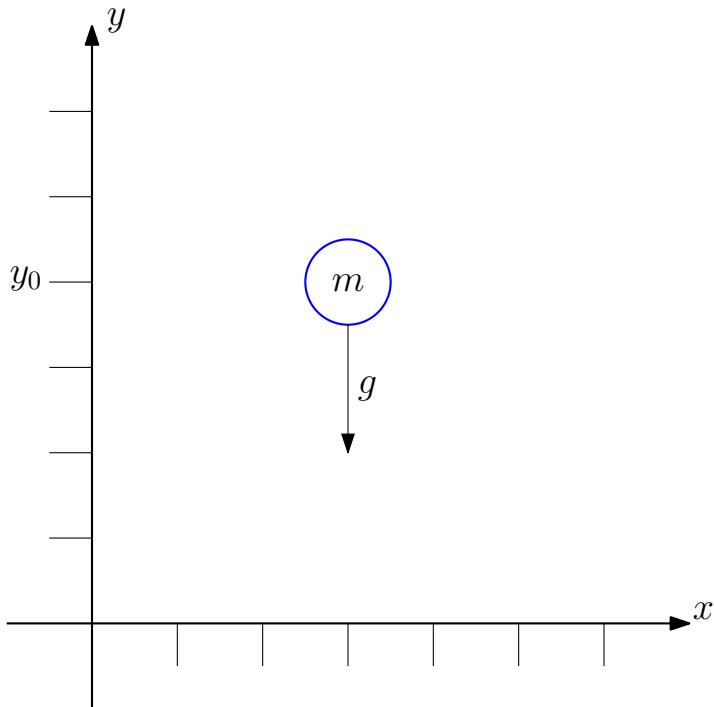


Figura 1.2: El modelo del cuerpo en caída libre, con su sistema de coordenadas.

de caída libre tendrá una solución matemática analítica y hasta cierto punto fácil de obtener. Hay muchas maneras de derivar estas leyes, más adelante cuando se unan todos los modelos se entenderá por qué elegí explicarla de esta manera y no de una manera más natural.

Empiezo planteando las condiciones del modelo: suponemos que se trata de un objeto con masa m , que se encuentra al iniciar su movimiento a una altura y_0 , que en el momento de iniciar su movimiento tiene una velocidad inicial v_0 y que durante todo su movimiento sólo va a actuar sobre él una fuerza constante F (Figura 1.2), que le imprime una aceleración constante a y que esta fuerza es precisamente la fuerza de gravedad que ejerce la tierra sobre él, es decir: $a = g$.

Además por seguir la convención usual, se moverá en un sistema coordenado derecho, es decir las direcciones positivas serán derecha en el eje x , y arriba en el eje y , por ende los movimientos hacia abajo y a la izquierda serán negativos, respecto a cada uno de los ejes. Además sabemos que la gravedad sólo afecta el movimiento vertical del cuerpo, es decir sólo actúa sobre el eje y . Entonces la segunda ley de Newton nos dice que:

$$F = ma \quad (1.1)$$

Pero nosotros sabemos que la aceleración a , es la derivada de la velocidad v , respecto al tiempo t . Y que la velocidad es a su vez la derivada de la posición y respecto al tiempo t , por lo que podemos escribir la ley de Newton como:

$$\frac{d^2y}{dt^2} = \frac{1}{m}F(t) \quad (1.2)$$

Además se sabe que tenemos dos condiciones iniciales, para nuestro problema:

$$y(0) = y_0$$

$$v(0) = v_0$$

Tenemos entonces una ecuación diferencial ordinaria, de segundo orden, con dos condiciones iniciales, por lo tanto podemos encontrar una solución única.

Esta ecuación es en particular, fácil de resolver debido a que la fuerza F es constante.

Primero sabemos que $F(t) = -mg$, el signo es porque la gravedad actúa hacia abajo. Y sustituyendo en la ecuación:

$$\frac{d^2y}{dt^2} = \left(\frac{1}{m}\right)(-mg) \quad (1.3)$$

Aquí vemos por qué Galileo encontró que la caída libre *no* depende de la masa del objeto.

Si integramos la ecuación respecto a t .

$$\frac{dy}{dt} = -gt + c_2$$

Donde c_2 es una constante de integración, que puede encontrarse de la primera condición inicial.

$$\frac{dy}{dt} = -gt + v_0$$

Podemos volver a integrar la ecuación respecto a t y volver a obtener la nueva constante c_1 de las condiciones iniciales.

$$\begin{aligned} y(t) &= -\frac{gt^2}{2} + v_0 t + c_1 \\ &= -\frac{gt^2}{2} + v_0 t + y_0 \end{aligned} \tag{1.4}$$

Que es la función que estábamos buscando, esta función nos dice cual es el valor de la coordenada y del objeto cuando ha transcurrido un tiempo t . Esto es precisamente lo que necesitamos saber para hacer una animación: una función que nos diga cómo se mueve algo conforme pasa el tiempo. También cabe señalar que en nuestro caso de la caída libre $v_0 = 0$, recordemos que el cuerpo se dejó caer, por lo que nuestra función se simplifica. Y se puede simplificar aún más, si escogemos un sistema de referencia tal que el movimiento del objeto empiece en el origen de dicho sistema es decir $y_0 = 0$. La forma más simple de nuestra función es entonces:

$$y(t) = -\frac{gt^2}{2} \tag{1.5}$$

Quizás este sistema puede parecer un poco simple, y en efecto lo es, en el sentido que pudo resolverse sin mayor problema y de una manera analítica, esto se debió a que la aceleración, y por ende la fuerza se supuso constante. El valor de este modelo radica más bien en la existencia de una solución analítica.

1.2 El sistema masa-resorte

Para explicar esta situación se partirá desde lo más simple a lo más complejo. El modelo más sencillo se conoce como un *oscilador armónico simple*. Después se tratará de agregar una segunda fuerza que servirá como un amortiguador, este nuevo modelo se conoce como *oscilador armónico amortiguado*. Por último este modelo se generalizará a un

sistema en tres dimensiones y con vectores y finalmente derivaremos la fórmula que ocuparemos durante el resto del trabajo.

La mecánica de vibraciones se debe al físico inglés Robert Hooke, la vida de este científico es muy interesante por diversos motivos.¹ Es uno de los científicos experimentales más importantes de la historia, y sus estudios abarcan campos muy diversos como la biología, la medicina, la cronometría, la física planetaria, la microscopía, la náutica y la arquitectura. Estuvo en disputas con Newton por el descubrimiento de la ley de la gravitación universal.

Aquí estamos interesados en uno de sus descubrimientos en elasticidad y que es conocido como *Ley de Hooke*. Que nos dice:

La fuerza restauradora ejercida por un resorte es linealmente proporcional al desplazamiento del resorte desde su posición de reposo y está dirigida hacia esa misma posición².

1.2.1 Oscilador armónico simple

Imagínese que se tiene una masa sujetada por un resorte sobre la que no actúa ninguna fuerza adicional. Por el momento vamos a suponer que no hay fricción del medio, presumiblemente el aire y que no está sujeta a la fuerza de gravedad. Por estas condiciones ideales la masa no tendrá pérdida de energía, es decir se moverá por siempre.

De nuevo la misma idea que antes; ocuparemos la ecuación de Newton y lo único que cambiará en nuestro modelo será la expresión para calcular la fuerza, en vez de la fuerza de gravedad, se modelará la fuerza del resorte.

Imaginemos el resorte; los resortes se estiran y se comprimen cuando se les aplica una fuerza. Siempre que pasa esto, el resorte en respuesta ejerce una fuerza restauradora, esta fuerza siempre lucha por devolver al resorte a su punto original de reposo es decir a donde no está ni comprimido ni estirado. Es entonces que la ley de Hooke para los resortes nos da una manera de calcular esta fuerza restauradora F_s .

¹Una biografía de Hooke se puede encontrar en la [wikipedia](#) en español; una biografía aun más completa, en [MacTutor](#)

²En el libro [2] en el apartado dedicado a Modelación por medio de Sistemas, se enuncia de esta manera, ignoro si Hooke pensó en resortes al enunciar su ley, o se le descubrió esta aplicación en un momento posterior.

Empecemos a plantear las condiciones del modelo, supongamos que es una masa unida por un resorte a un objeto inamovible, una pared por ejemplo y que todo el sistema descansa en el piso, para que no lo afecte la gravedad, y este piso no le da ningún tipo de fricción. En estas condiciones el único movimiento permitido para la masa será el horizontal y a la derecha, de nuevo por convención diremos que es a la dirección positiva del eje x . La Figura 1.3 ilustra la situación aquí descrita.

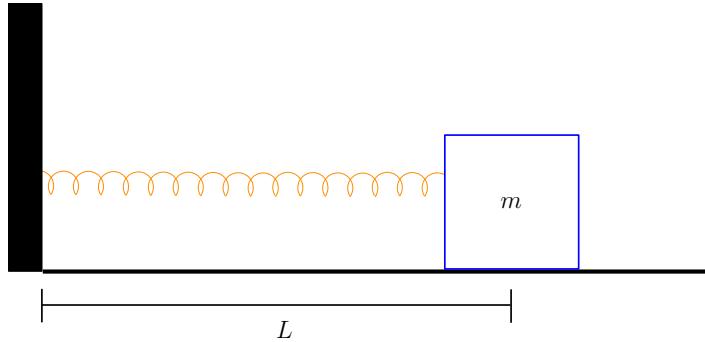


Figura 1.3: Una masa que se mueve solamente afectada por la fuerza de un resorte.

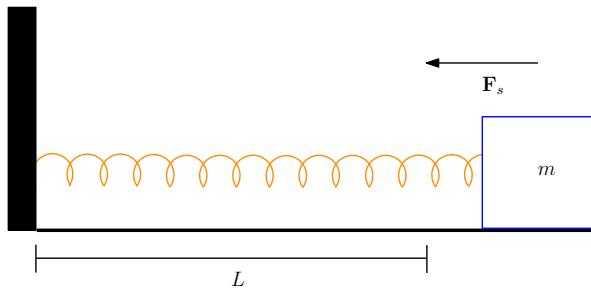
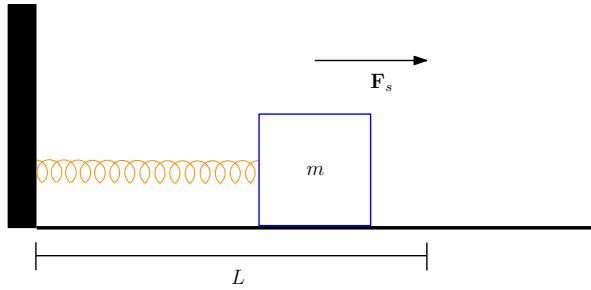
Se trata de modelar el movimiento de la masa a lo largo del tiempo t . La masa del cuerpo de nuevo la denotaremos como m y el origen del sistema será el punto donde el resorte se une a la pared. Supongamos también que cuando el resorte está en reposo tiene un largo L . La expresión que nos dice cual es la fuerza del resorte, que denotaremos como ya se dijo F_s , para un estiramiento razonablemente pequeño es la *ley de Hooke para los resortes* y se escribe como sigue:

$$F_s = -k_s(x - L) \quad (1.6)$$

Esta expresión básicamente nos dice que la fuerza es proporcional a la distancia de la masa a su punto de equilibrio donde el resorte está en reposo. La nomenclatura F_s se debe a que la mayoría de la bibliografía consultada está en inglés y traté de ser consistente con ella, ésta es la fuerza del resorte o *spring force*.

Se observa en esta expresión que cuando el resorte está estirado, es decir, su largo x es mayor a L la fuerza del resorte se vuelve negativa y por ende jala a la masa a su punto de reposo. Como se ve en la Figura 1.4a.

En el caso que el resorte esté compreso (Figura 1.4b), es decir, $x < L$, el resorte empuja

(a) Cuando $x > L$, el resorte jala la masa.(b) Cuando $x < L$, el resorte empuja la masa**Figura 1.4:** F_s trata de llevar el sistema a $x = L$

el cuerpo para que alcance la posición de reposo y de nuevo lo hace con una fuerza proporcional a la distancia que esté alejado del cuerpo de este punto.

También se observa que cuando el largo del resorte es $x = L$ el resorte está en reposo y por lo tanto la fuerza se hace cero, éste es el caso que se ilustró en la Figura 1.3.

Esta expresión para la fuerza se vuelve a aplicar a la ecuación de la ley de Newton, quedando la siguiente ecuación diferencial.

$$\begin{aligned}
 m \frac{d^2x}{dt^2} &= F(t) \\
 &= F_s \\
 &= -k_s(x - L) \\
 \frac{d^2x}{dt^2} + \frac{k_s}{m}x &= \frac{k_s L}{m}
 \end{aligned} \tag{1.7}$$

Esta ecuación de nuevo tendrá solución única si se especifican un par de condiciones iniciales de la forma $x(0) = x_0$ y $v(0) = v_0$. La teoría de ecuaciones diferenciales nos dice que esta solución se puede encontrar, mediante la suma de una solución particular x_p y una solución general de la ecuación diferencial homogénea asociada al problema

x_h . Es decir $x(t) = x_p + x_h$.

Por simple inspección se puede ver que $x(t) = L$ es una solución que nos puede servir como x_p y afortunadamente la ecuación homogénea es una *ecuación diferencial lineal ordinaria de coeficientes constantes* por lo que es relativamente simple de resolver. Además recuérdese que las constantes k_s , m y L , son todas positivas por lo que una solución es:

$$x_h = c_1 \cos \sqrt{\frac{k_s}{m}} t + c_2 \sin \sqrt{\frac{k_s}{m}} t \quad (1.8)$$

Con estas dos soluciones tenemos:

$$\begin{aligned} x(t) &= x_p + x_h \\ &= c_1 \cos \left(\sqrt{\frac{k_s}{m}} t \right) + c_2 \sin \left(\sqrt{\frac{k_s}{m}} t \right) + L \end{aligned} \quad (1.9)$$

Y podemos determinar las dos constantes de integración c_1 y c_2 de las condiciones iniciales.

Para este problema en particular la solución que estamos buscando es:

$$x(t) = (x_0 - L) \cos \left(\sqrt{\frac{k_s}{m}} t \right) + \left(v_0 \sqrt{\frac{m}{k_s}} \right) \sin \left(\sqrt{\frac{k_s}{m}} t \right) + L \quad (1.10)$$

Como puede apreciarse en la solución el cuerpo nunca deja de moverse, su movimiento está expresado por una función periódica. Esto es debido a que en un medio ideal como éste el sistema no disipa energía por lo tanto, nunca cesa su movimiento, y el resorte se comprime y se estira de la misma manera por siempre.

1.2.2 Oscilador armónico amortiguado

Como es lógico las condiciones ideales antes tratadas son casi imposibles de alcanzar en la realidad, por lo que es necesario agregar a nuestro sistema una manera de perder energía.

Una manera simple de hacerlo es agregándole una resistencia por parte del medio, por ejemplo el aire, o el agua si todo el sistema se ha sumergido bajo ésta.

Esta nueva fuerza en el sistema, a menudo se supone directamente proporcional a la velocidad del cuerpo con respecto al medio en el que se encuentra y se modela con la siguiente ecuación.

$$F_d = -k_d \frac{dx}{dt} \quad (1.11)$$

El signo es negativo debido a que la resistencia del medio se opone a la velocidad. Es decir trata de detener la masa poco a poco. De nuevo una aclaración sobre la nomenclatura: se dice que está es la fuerza de amortiguación y se refiere a la palabra amortiguador en inglés: *damping*. Y entonces el nuevo modelo toma la forma:

$$\begin{aligned} m \frac{d^2x}{dt^2} &= F_s + F_d \\ &= -k_s(x - L) - k_d \frac{dx}{dt} \\ m \frac{d^2x}{dt^2} + k_d \frac{dx}{dt} + k_s x &= k_s L \end{aligned} \quad (1.12)$$

Una vez más sujeto a las mismas condiciones iniciales $x(0) = x_0$ y $v(0) = v_0$.

Esta ecuación se resuelve de una manera similar a la de la sección pasada, donde una solución particular x_p vuelve a ser $x_p = L$. Para encontrar una solución general se requiere resolver la ecuación homogénea asociada:

$$m \frac{d^2x}{dt^2} + k_d \frac{dx}{dt} + k_s x = 0 \quad (1.13)$$

De nuevo la teoría nos dice que esta ecuación tiene una solución de la forma:

$$x(t) = c_1 e^{n_1 t} + c_2 e^{n_2 t}$$

Donde las constantes n_1 y n_2 son soluciones de la ecuación algebraica en n :

$$mn^2 + k_d n + k_s = 0 \quad (1.14)$$

Dado que ésta es una ecuación de segundo grado, puede resolverse con la fórmula general. Además se puede obtener información de la solución analizando cómo es k_d^2 con respecto a $4mk_s$ (lo que vendría a ser b^2 con respecto a $4ac$). Esto nos lleva a tres casos: que n tenga dos valores reales diferentes, que n tenga un solo valor real y que n tenga un par de valores complejos conjugados.

Estos casos son analizados a detalle en [15] y por lo tanto lo único que diré es que los dos primeros casos nos llevan a resultados donde el amortiguador es tan fuerte que no deja a la masa oscilar ni siquiera una vez. Es decir en donde la fuerza del medio es muy superior a la fuerza del resorte.

El único caso que a nosotros nos interesa es precisamente aquel donde el amortiguador es débil, y deja oscilar a la masa unas cuantas veces antes de detenerla, este caso se da precisamente cuando se trata de dos valores complejos conjugados de n .

Este caso es cuando $k_d^2 < 4mk_s$ es decir que el discriminante de la ecuación es negativo, por lo que la solución es de la forma:

$$\begin{aligned} n_1 &= \alpha + \beta i \\ n_2 &= \alpha - \beta i \end{aligned}$$

En donde:

$$\begin{aligned} \alpha &= \frac{k_d}{2m} \\ \beta &= \frac{\sqrt{4mk_s - k_d^2}}{2m} \end{aligned}$$

Y por tanto la ecuación diferencial tiene una solución de la forma:

$$x(t) = e^{-\alpha t} (c_1 \cos \beta t + c_2 \sin \beta t) + L \quad (1.15)$$

De nuevo por medio de las condiciones iniciales se pueden obtener los valores de las constantes c_1 y c_2 , quedando de esta manera:

$$\begin{aligned} c_1 &= x_0 - L \\ c_2 &= \frac{v_0 + (x_0 - L)\alpha}{\beta} \end{aligned}$$

De la solución (1.15) es claro que cuando el tiempo $t \rightarrow \infty$, el factor $e^{-\alpha t}$ dominará y la masa se detendrá poco a poco.

A manera de ejemplo ilustrativo, podemos elegir un conjunto arbitrario de valores para los parámetros. Por ejemplo, si $m = \frac{1}{2}$, $k_d = \frac{1}{10}$, $k_s = \frac{9+1/100}{2}$, $L = 3$, $x_0 = 6$ y $v_0 = 9 - 3/10$. Entonces $\alpha = \frac{1}{10}$, $\beta = 3$, $c_1 = 3$ y $c_2 = 3$. Lo que conduce a la solución $x(t) = e^{\frac{-1}{10}t} (3 \cos t + 3 \sin t) + 3$, cuyo comportamiento se puede ver en la Figura 1.5.

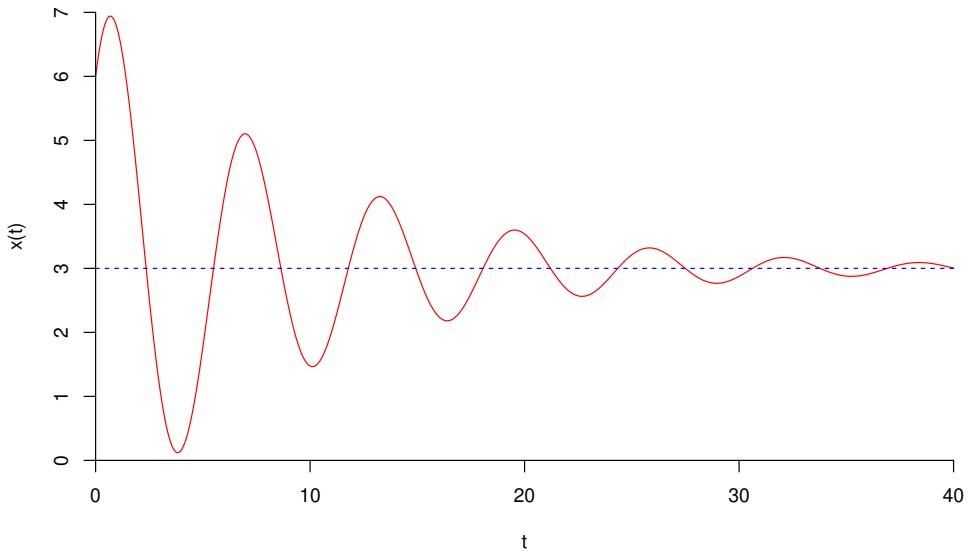


Figura 1.5: Gráfica de $x(t) = e^{\frac{-1}{10}t} (3 \cos t + 3 \sin t) + 3$. La línea discontinua en azul indica el estado de equilibrio: $x(t) = L$.

1.2.3 Masa-resorte en más de una dimensión

Tomaremos el modelo de la sección anterior; pero lo vamos a generalizar de manera que se pueda modelar este mismo fenómeno unidimensional solo que situado en un espacio tridimensional. Por medios geométricos trataré de aplicar las mismas fórmulas

de la sección pasada. Por simplicidad en los diagramas, voy a explicar el caso de dos dimensiones, sin embargo, como la explicación se hará con vectores, no se debe de tener problema en generalizar para tres dimensiones.

De aquí en adelante, para la notación matemática; usaré la convención de escribir los vectores en negritas y los escalares en fuente normal ³.

Primero vamos a considerar que hay dos puntos en el espacio que están unidos por un resorte, este resorte es la única fuerza que actúa sobre ellos. El resorte no tiene masa y ambos cuerpos tienen una masa m . Los puntos no están sujetos, es decir se mueven libremente por el espacio, y la única fuerza que actúa sobre ellos es el resorte.

Hay que notar que a diferencia de la sección pasada, donde el resorte estaba empotrado en la pared, aquí el resorte está unido a dos puntos sueltos, y por lo tanto aplica la misma fuerza sobre cada uno de los puntos. Esto se puede apreciar mejor en la Figura 1.6.

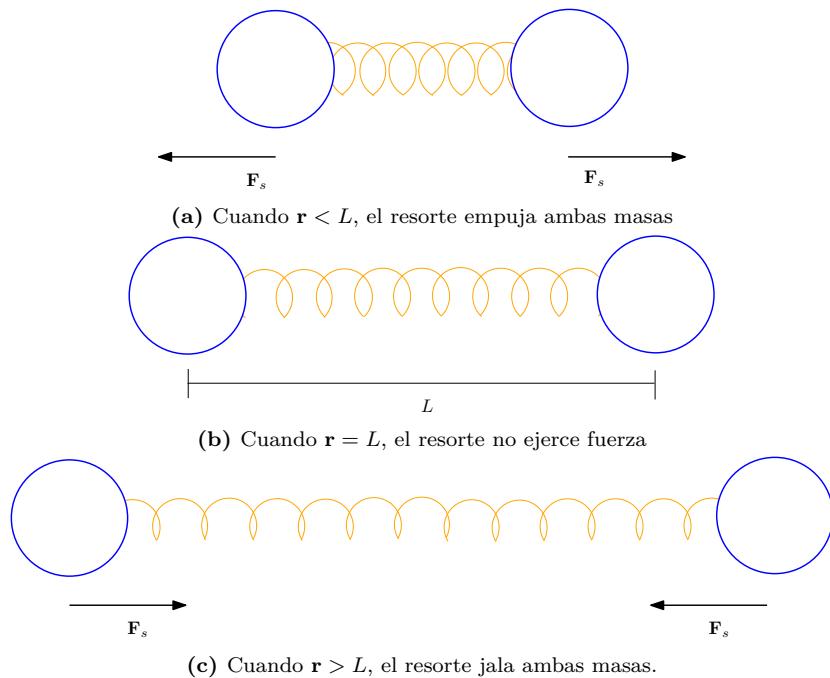


Figura 1.6: Como las masas están libres, el resorte aplica sobre cada una de ellas la misma fuerza, pero en sentido contrario.

Llamaré estos puntos a y b respectivamente, y vemos que una posible manera de localizarlos en el espacio es el vector de posición de cada uno de ellos, denotados por \mathbf{p}_a y

³Ésta es una convención muy aceptada en la literatura de las gráficas por computadora

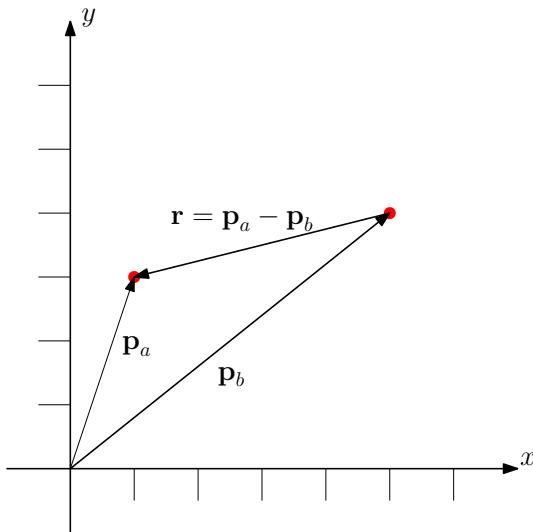


Figura 1.7: Podemos localizar puntos en el espacio por medio de sus vectores de posición.

\mathbf{p}_b .

1.2.3.1 La fuerza del resorte

Vamos a calcular la fuerza del resorte sobre el punto a ; la denotaré como \mathbf{F}_a , y por lo anterior ya se sabe que la fuerza que actúa sobre b es su negativo, es decir $\mathbf{F}_b = -\mathbf{F}_a$

Sabemos que una manera de encontrar el vector que une los puntos a y b es restando los vectores de posición de cada uno de ellos; a este nuevo vector lo llamaremos \mathbf{r} así que:

$$\mathbf{r} = \mathbf{p}_a - \mathbf{p}_b \quad (1.16)$$

Como se ve en la Figura 1.7 este vector empieza en la punta de \mathbf{p}_b y termina en la punta de \mathbf{p}_a . Entonces la distancia que hay de a a b es precisamente la norma de este vector \mathbf{r} , dicho de otra manera $\mathbf{r} = \mathbf{p}_a - \mathbf{p}_b$ y $\overline{ab} = |\mathbf{r}|$.

Ahora que sabemos la distancia de a a b , la podemos restar con el tamaño de resorte que es el escalar L . Si a este resultado lo multiplicamos por la constante del resorte k_s sabremos la magnitud de la fuerza del resorte, es decir calculando $k_s (|\mathbf{r}| - L)$.

Éste es un escalar, que representa la magnitud de la fuerza, pero como ésta última es

un vector, hace falta que le demos dirección, para hacerlo multiplico el escalar por el vector \mathbf{r} normalizado.

Analizaremos la acción de la fuerza. Si el factor $k_s (|\mathbf{r}| - L)$ es negativo, entonces $|\mathbf{r}| < L$, significa que el resorte está comprimido, luego entonces la fuerza debe empujar el punto a lejos del punto b , como ya dijimos que \mathbf{r} representa un vector que va de b a a , entonces la fuerza debe tener la dirección de este vector. Veamos el caso contrario, si $|\mathbf{r}| > L$ el resorte está estirado, $k_s (|\mathbf{r}| - L)$ es positivo y la fuerza debe jalar el punto a en dirección de b , es decir en la dirección contraria que el vector \mathbf{r} . De estas dos observaciones nos damos cuenta de que el signo de $k_s (|\mathbf{r}| - L)$ debe ser negativo, es decir:

$$\mathbf{F}_a = -k_s (|\mathbf{r}| - L) \frac{\mathbf{r}}{|\mathbf{r}|} \quad (1.17)$$

Como se dijo, sólo se ha calculado la fuerza que actúa sobre el punto a , sin embargo por la simetría del sistema y que ambos puntos están sueltos en el espacio, la fuerza sobre el punto b , es equivalente pero en sentido contrario, es decir:

$$\mathbf{F}_b = -\mathbf{F}_a$$

1.2.3.2 La fuerza del amortiguador

En la sección pasada sólo modelamos la fuerza de un resorte que en esencia presenta la misma carencia del primer resorte que analizamos: le falta una manera de perder energía. Ya habíamos analizado una manera eficaz de hacer que el oscilador disipe energía, sin embargo esto se debía al medio. Aquí la idea es un poco diferente, vamos a pensar que esta resistencia no está presente en el medio, sino en el mismo resorte en sí; es decir nuestro resorte además de ser tal será un amortiguador, y la resistencia actuará por lo tanto en la misma línea de acción del resorte.

Ya dijimos anteriormente que esta resistencia la vamos a suponer proporcional a la velocidad. Y con la misma idea que antes vamos a agregarla al modelo anterior. Ya sabemos que tenemos una manera de localizar a los puntos a y b en el espacio, su vector de posición, sin embargo aún no tenemos una manera de saber la velocidad de los puntos en el espacio, por ello vamos a crear dos vectores \mathbf{v}_a y \mathbf{v}_b , que contienen la velocidad a la que se mueven estos puntos respectivamente.

Procedamos a calcular la fuerza que actúa sobre el punto a . Estas fuerzas son: la fuerza anterior debida al resorte que los une, y una nueva fuerza ahora debida al amortiguador.

$$\mathbf{F}_a = \mathbf{F}_s + \mathbf{F}_d \quad (1.18)$$

Donde \mathbf{F}_s es precisamente la expresión (1.17). Así que sólo nos falta calcular \mathbf{F}_d . Esta fuerza, como ya se dijo, es proporcional a la velocidad a la que se mueve un punto respecto al otro ($\mathbf{v}_a - \mathbf{v}_b$) con una constante k_d , y además actúa en el sentido contrario al de la velocidad, es decir F_d es proporcional a $-k_d(\mathbf{v}_a - \mathbf{v}_b)$.

Ahora hay que notar también que la posición de un punto no necesariamente tiene que ver con su velocidad, es decir un punto en una misma posición se podría mover bien a una velocidad \mathbf{v} u a otra muy diferente, sin tener que cambiar su vector de posición.

Como ya dijimos que para nosotros el amortiguador está en el resorte, es necesario entonces trasladar esta fuerza a donde está el resorte, es decir a $\mathbf{r} = \mathbf{p}_a - \mathbf{p}_b$. Para hacer esto vamos a ocupar la conocida forma de la proyección de un vector sobre otro.

Sabemos que un vector \mathbf{A} se puede proyectar sobre un vector \mathbf{B} y se denota como $\text{Proy}_{\mathbf{B}}\mathbf{A}$ podemos obtener su magnitud con la siguiente fórmula:

$$|\text{Proy}_{\mathbf{B}}\mathbf{A}| = \frac{\mathbf{A} \cdot \mathbf{B}}{|\mathbf{B}|} \quad (1.19)$$

Ahora entonces calculemos la magnitud de la proyección del vector $(\mathbf{v}_a - \mathbf{v}_b)$ sobre el vector donde está el resorte, es decir sobre el vector $\mathbf{r} = \mathbf{p}_a - \mathbf{p}_b$. Quedando de esta manera:

$$|\text{Proy}_{\mathbf{r}}\mathbf{v}| = \left[\frac{(\mathbf{v}_a - \mathbf{v}_b) \cdot (\mathbf{p}_a - \mathbf{p}_b)}{|\mathbf{p}_a - \mathbf{p}_b|} \right] \quad (1.20)$$

Sin embargo, esta cantidad es un escalar así que necesitamos multiplicarla por un vector unitario para darle dirección. Como esta fuerza es debida al *resorte-amortiguador* y está en su misma línea de acción, la multiplicamos por \mathbf{r} normalizado. Quedando la expresión final para \mathbf{F}_d :

$$\mathbf{F}_d = -k_d \left[\frac{(\mathbf{v}_a - \mathbf{v}_b) \cdot (\mathbf{p}_a - \mathbf{p}_b)}{|\mathbf{p}_a - \mathbf{p}_b|} \right] \left[\frac{\mathbf{p}_a - \mathbf{p}_b}{|\mathbf{p}_a - \mathbf{p}_b|} \right] \quad (1.21)$$

Como ya expliqué antes ésta es sólo la fuerza del amortiguador así que la expresión de fuerza para este *resorte-amortiguador* (reemplazando $\mathbf{r} = \mathbf{p}_a - \mathbf{p}_b$ y $\mathbf{v} = \mathbf{v}_a - \mathbf{v}_b$ para no hacer muy compleja la notación) es la siguiente:

$$\begin{aligned} \mathbf{F}_a &= \mathbf{F}_s + \mathbf{F}_d \\ \mathbf{F}_a &= -k_s (|\mathbf{r}| - L) \frac{\mathbf{r}}{|\mathbf{r}|} - k_d \left[\frac{\mathbf{v} \cdot \mathbf{r}}{|\mathbf{r}|} \right] \left[\frac{\mathbf{r}}{|\mathbf{r}|} \right] \\ \mathbf{F}_a &= - \left\{ k_s (|\mathbf{r}| - L) + k_d \left[\frac{\mathbf{v} \cdot \mathbf{r}}{|\mathbf{r}|} \right] \right\} \left[\frac{\mathbf{r}}{|\mathbf{r}|} \right] \\ \mathbf{F}_b &= -\mathbf{F}_a \end{aligned} \quad (1.22)$$

1.3 El modelo del gas ideal

Aquí se va a tratar de agregar al modelo, una fuerza de presión. La idea de esta fuerza es tomada del artículo de Matika [11], y se basa en implementar un modelo simple de gas. Este modelo cuenta con la ventaja de la sencillez, tanto en su implementación como en la rapidez de su ejecución, lo que lo hace un modelo ideal para una animación en tiempo real.

Vamos a empezar por imaginar un cuerpo cerrado en tres dimensiones, por ejemplo una esfera o un cubo, luego vamos a imaginar que la envolvente de ese cuerpo, es decir la superficie que lo cierra, es una superficie elástica, que se puede estirar y deformar, y por último supongamos que dentro de ese cuerpo, en vez de haber un vacío, hay un gas; ésta es la idea principal del modelo que trataremos de ahora en adelante, y es el motivo de estudio de este trabajo.

En las dos secciones anteriores hemos tratado los modelos que nos pueden servir como la envolvente de este cuerpo (podría ser una envolvente formada por una membrana de puntos unidos por resortes), pero hablaré más de esto en el siguiente capítulo. Así que sólo falta tratar el modelo de un gas por separado, para terminar nuestro tratamiento de modelos simples. El modelo que vamos a estudiar es el del *gas ideal*.

1.3.1 La termodinámica

Habíamos estado hablando de leyes de mecánica, sin embargo este modelo pertenece a una rama diferente de la física, la *termodinámica*. La termodinámica es la rama de la física que estudia los efectos de los cambios de temperatura, calor y presión, en un sistema, por medio de métodos estadísticos.⁴

El estudio de la termodinámica empieza con el descubrimiento de las maneras de medir el calor, es decir, con la invención de los primeros termómetros, fue así como se notó que existen sistemas que presentan cambios cualitativos, cuando cambian su temperatura.

Uno de estos sistemas es el gas, y por lo tanto existen algunos modelos para estudiar las propiedades de los gases. El más simple de estos modelos es el modelo del gas ideal, que se define por medio de la *ley de los gases ideales*, que no es otra cosa que la ecuación de estado de un gas ideal.

1.3.2 Un modelo simple

Al principio, para poder estudiar un fenómeno del cual no se conocen muchas cosas, es común relajar muchas de sus propiedades e idealizar muchas situaciones, es decir construir modelos simples, en este caso nos abocamos al primer modelo que se construyó para un gas, y por lo mismo está sujeto a muchas suposiciones que en la realidad nunca son alcanzadas por ningún gas, de ahí que lo llamaran *gas ideal*.

En física y en química, una ecuación de estado es una ecuación que describe el estado de agregación de la materia en función de ciertos parámetros, es decir determina la relación matemática entre dos o más funciones de estado asociadas con la materia.

En nuestro caso queremos describir el estado de un gas en función de propiedades macroscópicas del mismo. Estas propiedades son: la temperatura, el volumen, la presión y el número de moléculas.

Por medio de resultados experimentales se determinó la primera ecuación de estado, una explicación de cómo se hicieron estos experimentos y se llegó a ella se puede encontrar en [14], pero a nosotros nos bastará con conocerla:

⁴En esta definición de *termodinámica* comúnmente se aplican modelos más complejos que el usado aquí, que sí requieren de métodos estadísticos, pero de hecho nuestro modelo es en esencia determinista.

$$\begin{aligned} PV &= nRT \\ P &= \frac{1}{V}nRT \end{aligned} \tag{1.23}$$

En donde P es la presión del gas, V es el volumen del gas, n es el número de moles, R es una constante llamada constante del gas ideal y T es la temperatura. La constante R es la multiplicación de la constante de Boltzman y el número de Avogadro ⁵. Las unidades de esta constante son Joule (J) sobre mol (mol) Kelvin (K).

$$R = N_A k = 8.3145 \frac{J}{mol \cdot K}$$

Esta ecuación de estado es un aproximación bastante buena cuando se trata de gases a bajas densidades, y la experimentación nos muestra que en estas condiciones *todos* los gases tienden a comportarse como este gas ideal.

1.3.3 La presión de un gas

La presión no es en sí una fuerza sino es una fuerza aplicada en una unidad de área, y por eso se mide en una unidad llamada *Pascals* (Pa) y no en Newtons (N) que es la unidad en que se mide la fuerza. Entonces, si \mathbf{P} es la presión, \mathbf{F} es una fuerza y A el área donde se está aplicando esta fuerza:

$$\mathbf{P} = \frac{\mathbf{F}}{A}$$

De tal manera que si nosotros queremos una *fuerza de presión*, para poder acumularla en la ecuación de Newton; como lo hicimos con las fuerzas anteriores necesitamos multiplicarla por el área de aplicación. Dicho de otra manera:

$$\mathbf{F}_p = \mathbf{P} \cdot A$$

⁵Esta constante universal aparece en varias ecuaciones importantes de la termodinámica, es equivalente a la constante de Boltzman pero expresada en unidades de energía.

Donde \mathbf{F}_p es la fuerza de presión que estamos buscando, A es el área de aplicación y \mathbf{P} es el vector que representa la presión. Ya se dijo que la magnitud de la presión puede calcularse con la ecuación de estado, pero no se ha dicho qué dirección lleva. La presión actúa sobre la superficie de la cara del cuerpo, y es siempre de normal a esta cara, de manera que la presión es:

$$\mathbf{P} = P \cdot \mathbf{n}$$

En donde P es el escalar calculado con la ecuación de estado y \mathbf{n} es un vector unitario normal a la cara sobre la que se está calculando la presión.

Para nuestro modelo se debe tomar en cuenta que sólo nos interesa calcular la fuerza de presión, de esta manera se procede de la siguiente manera.

$$P = \frac{1}{V} nRT$$

De esta manera \mathbf{P} es:

$$\mathbf{P} = \frac{1}{V} nRT [\mathbf{n}]$$

Y este vector \mathbf{P} se usará a su vez, para obtener el vector fuerza, es decir:

$$\mathbf{F}_p = \frac{1}{V} nRT [\mathbf{n}] A$$

Reacomodando esta ecuación, se tiene:

$$\mathbf{F}_p = \left[\frac{1}{V} A nRT \right] \mathbf{n}$$

Donde para *nuestro* modelo el escalar nRT es un parámetro a determinarse experimentalmente, está formado por la temperatura que consideramos constante, el número molar del gas, que debe depender del gas y por la constante del gas ideal. Así que los tres son agrupados en una sola constante, que nosotros por ser consistentes con nuestra nomenclatura llamaremos k_g

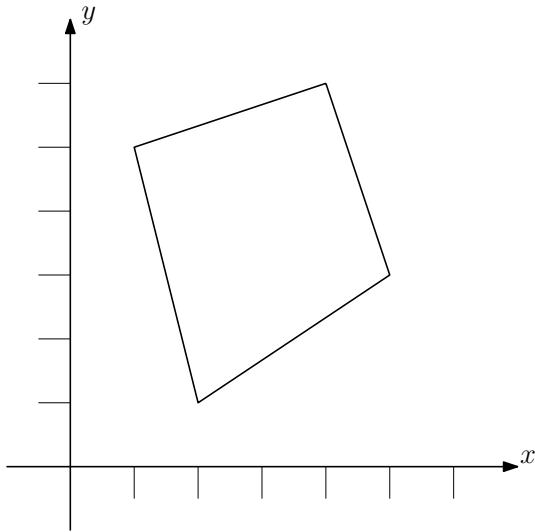


Figura 1.8: La figura cerrada para la cual se harán los cálculos de presión.

De manera que:

$$\mathbf{F}_p = \left[\frac{1}{V} A k_g \right] \mathbf{n} \quad (1.24)$$

Donde \mathbf{F}_p es la fuerza que deseamos acumular en cada partícula, V es el volumen total de nuestro cuerpo cerrado, A es el área de la cara sobre la que estamos calculando la presión, \mathbf{n} es un vector unitario normal a esta cara y k_g es una constante positiva a ser determinada experimentalmente. Es la ecuación que nos servirá en nuestro modelo.

1.3.4 Un ejemplo ilustrativo

Supongamos que se tiene una caja cerrada cuya *tapa* tiene el largo y el ancho mostrado en la Figura 1.8 la caja tiene una altura igual a 1, es decir, $h = 1$. Esta caja tiene las tapas fijas a lo alto, por lo que la fuerza de un gas sólo la puede deformar en sus lados. Calculemos ahora la fuerza debida a la presión de un gas que esté dentro de esta caja.

Vamos a calcular la fuerza correspondiente a la presión, por medio de la ecuación (1.24). Como la caja es de altura unitaria, el volumen V que es igual al área de su tapa por la altura equivale simplemente al área de su tapa. Por la misma razón las áreas de las caras laterales de la caja equivalen a la longitud del lado de la tapa en el que se encuentran.

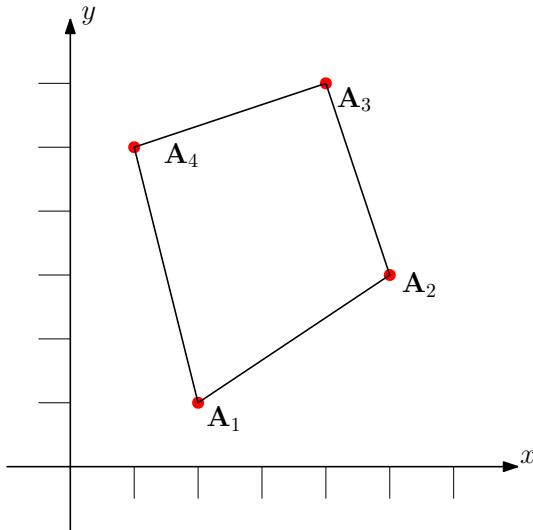


Figura 1.9: Figura con los puntos que representan las esquinas identificados y localizados.

Para calcular la fuerza de presión en este caso se necesita saber entonces: la longitud de cada uno de los lados del cuadrilátero, un vector normal a cada una de sus lados y su área. Por suerte para nosotros lo único que necesitamos para conocer todos estos datos, es ubicar la posición de cada una de sus esquinas.

Identificaremos cada una de las esquinas \mathbf{A}_i por medio de sus coordenadas x_i y y_i , dicho de otra manera $\mathbf{A}_i = (x_i, y_i)$. Esta situación queda determinada por la Figura 1.9.

De donde podemos conocer, gracias al sistema de referencia, las coordenadas de cada uno de los puntos en donde se ubican las esquinas:

$$\begin{aligned}\mathbf{A}_1 &= (x_1, y_1) = (2, 1) \\ \mathbf{A}_2 &= (x_2, y_2) = (5, 3) \\ \mathbf{A}_3 &= (x_3, y_3) = (4, 5) \\ \mathbf{A}_4 &= (x_4, y_4) = (1, 4)\end{aligned}$$

Una vez conocidos estos datos podemos calcular la longitud de cada lado simplemente obteniendo la distancia entre dos puntos.

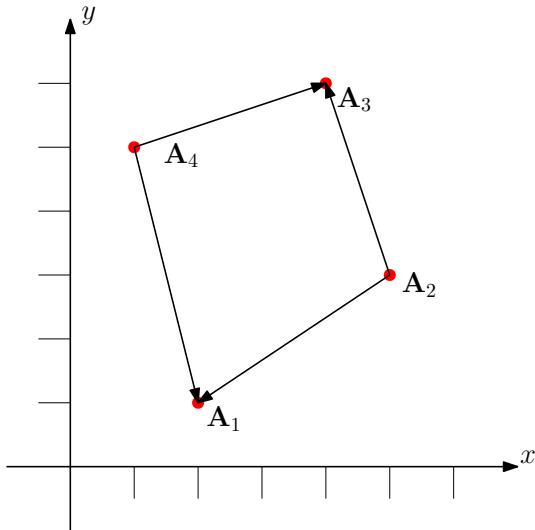


Figura 1.10: Aquí se pueden ver los vectores que representan cada uno de los lados, y que son calculados por la resta de los vectores de posición de los puntos.

$$\overline{\mathbf{A}_1\mathbf{A}_2} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} = \sqrt{13}$$

$$\overline{\mathbf{A}_2\mathbf{A}_3} = \sqrt{(x_2 - x_3)^2 + (y_2 - y_3)^2} = \sqrt{5}$$

$$\overline{\mathbf{A}_3\mathbf{A}_4} = \sqrt{(x_3 - x_4)^2 + (y_3 - y_4)^2} = \sqrt{10}$$

$$\overline{\mathbf{A}_4\mathbf{A}_1} = \sqrt{(x_4 - x_1)^2 + (y_4 - y_1)^2} = \sqrt{10}$$

También podemos calcular el área de la tapa con un sencillo truco, podemos trazar la diagonal que divide al cuadrilátero tal que pase por \mathbf{A}_1 y \mathbf{A}_3 , con lo que para calcular el área total del cuadrilátero basta con calcular el área de los dos triángulos $\triangle \mathbf{A}_1 \mathbf{A}_3 \mathbf{A}_4$ y $\triangle \mathbf{A}_1 \mathbf{A}_2 \mathbf{A}_3$ y luego sumar ambas áreas (ver la Figura 1.10).

Para calcular el área del triángulo $\triangle \mathbf{A}_1 \mathbf{A}_3 \mathbf{A}_4$ tomamos a \mathbf{A}_4 como origen y calculamos el vector que va de \mathbf{A}_4 a \mathbf{A}_1 , luego calculamos el vector que va de \mathbf{A}_4 a \mathbf{A}_3 , y por último calculamos el producto cruz, tomamos su norma y lo dividimos entre dos.⁶ De manera análoga para el siguiente triángulo:

⁶Este es un teorema muy conocido de la geometría vectorial y se puede ver en cualquier libro que incluya una introducción a los vectores, me imagino que no necesito citar alguna fuente en especial

El cálculo de los vectores es el siguiente:

$$\begin{aligned}\overrightarrow{A_4A_1} &= \mathbf{A}_1 - \mathbf{A}_4 = (1, -3) \\ \overrightarrow{A_4A_3} &= \mathbf{A}_3 - \mathbf{A}_4 = (3, 1) \\ \overrightarrow{A_2A_1} &= \mathbf{A}_1 - \mathbf{A}_2 = (-3, -2) \\ \overrightarrow{A_2A_3} &= \mathbf{A}_3 - \mathbf{A}_2 = (-1, 2)\end{aligned}$$

Y por lo explicado anteriormente el área total de la tapa A_T es:

$$\begin{aligned}A_T &= A_{\triangle \mathbf{A}_1 \mathbf{A}_3 \mathbf{A}_4} + A_{\triangle \mathbf{A}_1 \mathbf{A}_2 \mathbf{A}_3} \\ A &= \frac{1}{2} |\overrightarrow{\mathbf{A}_4 \mathbf{A}_1} \times \overrightarrow{\mathbf{A}_4 \mathbf{A}_3}| + \frac{1}{2} |\overrightarrow{\mathbf{A}_2 \mathbf{A}_1} \times \overrightarrow{\mathbf{A}_2 \mathbf{A}_3}| \\ A &= \frac{1}{2} |10| + \frac{1}{2} |-8| \\ A &= 9\end{aligned}$$

Ahora sólo nos falta calcular la fuerza \mathbf{F}_p y acumularla en cada una de las caras laterales de la caja. Para esto y sólo por simplicidad supongamos que escogemos $k_d = 10$ entonces el cálculo de la fuerza debe hacerse por separado para cada una de las caras laterales (que en nuestro caso equivalen a los lados de la tapa):

$$\begin{aligned}F_{\overline{\mathbf{A}_1 \mathbf{A}_2}} &= \frac{1}{V} \overline{\mathbf{A}_1 \mathbf{A}_2} k_g = \frac{1}{9} \sqrt{13} (10) \\ F_{\overline{\mathbf{A}_2 \mathbf{A}_3}} &= \frac{1}{V} \overline{\mathbf{A}_2 \mathbf{A}_3} k_g = \frac{1}{9} \sqrt{5} (10) \\ F_{\overline{\mathbf{A}_3 \mathbf{A}_4}} &= \frac{1}{V} \overline{\mathbf{A}_3 \mathbf{A}_4} k_g = \frac{1}{9} \sqrt{10} (10) \\ F_{\overline{\mathbf{A}_4 \mathbf{A}_1}} &= \frac{1}{V} \overline{\mathbf{A}_4 \mathbf{A}_1} k_g = \frac{1}{9} \sqrt{10} (10)\end{aligned}$$

Con lo que tenemos la magnitud de las fuerzas que actúan en cada cara, así que sólo falta calcular un vector unitario normal a la cara correspondiente para poder acumular su fuerza.

Afortunadamente, para calcular un vector normal en dos dimensiones, tenemos un mé-

todo sencillo, el producto punto de dos vectores normales entre sí debe de dar cero, así que si queremos un vector normal al vector $\mathbf{v} = (x, y)$ simplemente construimos uno tal que su producto punto con el primero siempre sea igual a cero, por ejemplo $\mathbf{v}^n = (-y, x)$. Con esto sabemos que el vector \mathbf{v}^n es siempre normal al vector \mathbf{v} , de hecho en dos dimensiones sólo este vector y sus múltiplos son normales a \mathbf{v} .

Sabiendo todo lo anterior, ya se puede acumular la fuerza correspondiente sobre cada cara. Por ejemplo, sobre la primera cara que está definida por el segmento que une a \mathbf{A}_1 y \mathbf{A}_2 . Como en el paso anterior ya se había calculado el vector $\overrightarrow{\mathbf{A}_2\mathbf{A}_1} = (-3, -2)$ que en sí representa a la cara, podemos calcular su vector normal con la fórmula anterior. Para esta cara el vector $\mathbf{n} = (2, -3)$ es el vector normal, para hacerlo unitario lo dividimos entre su norma $\frac{\mathbf{n}}{|\mathbf{n}|} = \frac{1}{\sqrt{13}}(2, -3)$. Y finalmente el vector fuerza que estamos buscando para esta cara en particular es:

$$\mathbf{F}_{\overline{\mathbf{A}_1\mathbf{A}_2}} = \frac{1}{V} \overline{\mathbf{A}_1\mathbf{A}_2} k_g \vec{n} = \frac{1}{9} \sqrt{13} (10) \frac{1}{\sqrt{13}} (2, -3) = \frac{10}{9} (2, -3)$$

Una consideración más que debe tenerse, al calcular el vector normal, pudiera darse el caso que éste dirija la fuerza al contrario de como se desea, es decir que la dirija, al centro del cuerpo en vez de hacia afuera. Como se ilustra en la Figura 1.11. En cuyo caso hay que multiplicar este vector normal por -1 . En general la mejor práctica es graficar el cuerpo para así poder calcular el vector normal correcto.

Por ejemplo, al calcular un vector normal para la segunda cara podríamos vernos tentados a repetir el cálculo, ocupando el vector que acabamos de calcular $\overrightarrow{\mathbf{A}_4\mathbf{A}_1}$ pero al ocupar la fórmula, nos daría $n = (3, 1)$ que en efecto es un vector normal a $\overrightarrow{\mathbf{A}_4\mathbf{A}_1}$ pero que apunta hacia el centro del cuerpo, y por lo tanto sería incorrecto ocuparlo para dirigir la fuerza de presión. El vector correcto para este caso es $-n = (-3, -1)$. Así que se debe tener cuidado al calcular el vector para dirigir la fuerza de presión.

El resto de las operaciones correspondientes al ejemplo se pueden ver en el Cuadro 1.1. En la primera columna de la tabla se muestra la cara a la que se estaba calculando la fuerza, después se muestra el valor escalar de la presión, luego el vector asociado a la cara, que ya habíamos obtenido, después el vector normal resultado del procedimiento antes explicado. Hay que tomar en cuenta que el vector podría ser que apunte hacia afuera o hacia adentro, y en la tabla ya fue corregido (multiplicándolo por el escalar -1). Y en la última columna aparecen los vectores de fuerza que se estaban buscando,

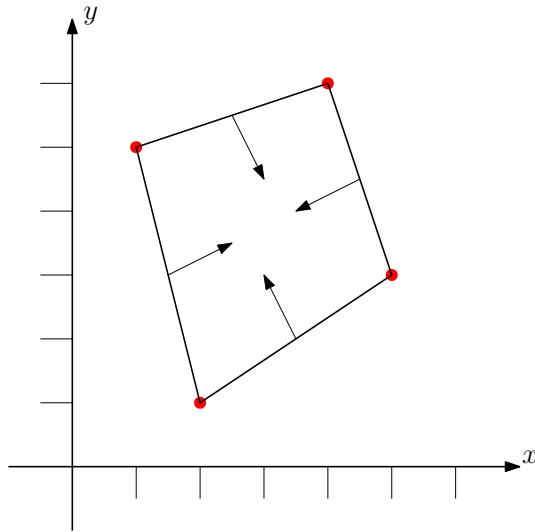


Figura 1.11: Aquí se pueden ver los vectores incorrectos para dirigir la presión pues apuntan al centro de la figura.

Cara	Fuerza de Presión	Vector de la cara	Vector Normal	Vector Fuerza
$\overline{A_1A_2}$	$\frac{1}{9} \cdot 10 \cdot \sqrt{13}$	$\overrightarrow{A_2A_1} = (-3, -2)$	$(\frac{2}{\sqrt{13}}, \frac{-3}{\sqrt{13}})$	$(\frac{20}{9}, \frac{-30}{9})$
$\overline{A_2A_3}$	$\frac{1}{9} \cdot 10 \cdot \sqrt{5}$	$\overrightarrow{A_2A_3} = (-1, 2)$	$(\frac{2}{\sqrt{5}}, \frac{1}{\sqrt{5}})$	$(\frac{20}{9}, \frac{10}{9})$
$\overline{A_3A_4}$	$\frac{1}{9} \cdot 10 \cdot \sqrt{10}$	$\overrightarrow{A_4A_3} = (3, 1)$	$(\frac{-1}{\sqrt{10}}, \frac{3}{\sqrt{10}})$	$(\frac{-10}{9}, \frac{30}{9})$
$\overline{A_1A_4}$	$\frac{1}{9} \cdot 10 \cdot \sqrt{10}$	$\overrightarrow{A_4A_1} = (1, -3)$	$(\frac{-3}{\sqrt{10}}, \frac{-1}{\sqrt{10}})$	$(\frac{-30}{7}, \frac{-10}{9})$

Cuadro 1.1: Cálculo de la Fuerza de Presión, para el ejemplo

y son los que se requieren para poder acumularse en la ecuación de Newton.

La fuerza de presión es mucho más compleja de calcular que las fuerzas debidas a la gravedad y a los *resortes-amortiguadores*. La principal dificultad es que debe tratar con la geometría del cuerpo para calcularla, además es necesario conocer el volumen total del cuerpo, y por cada cara que se acumule se debe de conocer: el área de la misma y un vector normal, por lo que a diferencia de los casos anteriores no hay una manera general de enunciar las fórmulas sino que debe analizarse el caso particular de la geometría del cuerpo que se esté tratando.

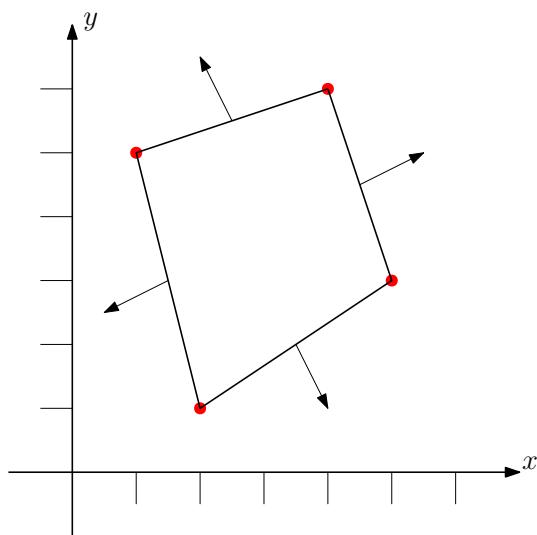


Figura 1.12: Así es como debe de dirigirse la fuerza de presión. Obsérvese que cada esquina debe acomullar la fuerza de las dos caras laterales que está uniendo.

Capítulo 2

Construcción del algoritmo de simulación

En este capítulo nos encargaremos de presentar el algoritmo propuesto por Matyka en [11], que es explicado en más detalle en [10]. Este modelo no es más que la unión de los modelos del capítulo anterior. Además definiremos el experimento que deseamos realizar para probar el funcionamiento del modelo.

Más adelante en este mismo capítulo, explicaremos cómo se pueden enfrentar ciertos detalles sobre el algoritmo de Matika.

Primero veremos cómo se puede calcular el volumen de un cuerpo en tres dimensiones, explicaremos cómo se puede tener una idea general, y por qué para ciertas geometrías del cuerpo, se prestan mejor ciertos algoritmos.

Luego nos enfrentaremos al problema de cómo integrar la ecuación de Newton, con métodos numéricos, analizando dos alternativas, y se presentan las que se consideran las mejores maneras de implementar el integrador en código.

Por último analizaremos el problema más difícil de la implementación: las colisiones. Esto se analiza en dos partes, la manera como se detectan y la respuesta de las mismas.

2.1 Diseño del experimento

Necesitamos una simulación gráfica (animación) y el principal objetivo será hacer que se vea real.

Se quiere modelar un cuerpo neumático. Para hacerlo partimos de un cuerpo flexible y le ponemos un fluido dentro, en este caso el gas, además queremos una manera de probar cómo es la interacción de éste con otros objetos de la escena.

Con base en esos requerimientos, diseñe la siguiente situación: se modela una caja cerrada, en la cual su cara superior tiene la característica de ser formada como un cuerpo flexible, las otras cinco caras son rígidas. Dentro de esta caja se pone el fluido (gas), de manera que ahora tenemos un cuerpo neumático. Para probar su interacción con otro cuerpo se le deja caer sobre su cara flexible un cuerpo sólido (en este caso una esfera) y se observa el resultado.

Además para poder hacer varios tipos de pruebas se desea que la animación tenga cierta interactividad, es decir que el usuario pueda, en tiempo de ejecución, cambiar algunas opciones tanto de visualización como también algunos parámetros físicos.

Esta simulación es el objetivo principal del trabajo, y el siguiente capítulo está dedicado a cómo se construyó, mientras que en este capítulo se hace énfasis en los detalles teóricos necesarios para escribir el programa.

2.2 Simulaciones gráficas basadas en física

Lo primero que debemos de hacer es exponer las condiciones de nuestra simulación de cuerpo flexible. La situación física que deseamos modelar es la de un cuerpo flexible, hueco que contiene un gas. Por ejemplo, imagínese un globo, un colchón de aire, la llanta de un automóvil o una burbuja. El comportamiento en el que estamos interesados es en un comportamiento cualitativamente parecido al del fenómeno, es decir que al momento de graficarlo y hacer una animación debe verse *parecido* al sistema real.

El valor de poder graficar de manera realista un modelo como éste, radica en el hecho de que hay muchas cosas que se comportan de manera parecida, y al momento de animar una escena donde hay varios de estos objetos, un modelo físicamente más realista sería

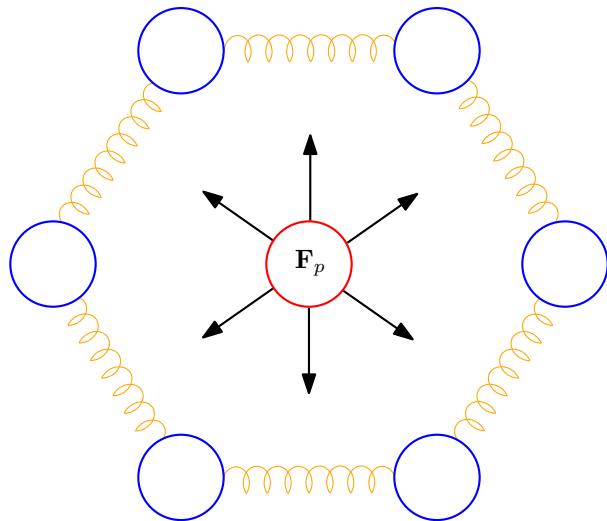


Figura 2.1: Un ejemplo de un modelo de masa resorte, con presión. Se trata de un hexágono cerrado, y \mathbf{F}_p representa la fuerza debida a la presión.

muy costoso en tiempo de ejecución.

La forma de atacar este problema será hacerlo con un modelo simple. Imagínese un cuerpo formado por varios puntos unidos entre ellos por resortes.¹ Ahora imagínese que a un cuerpo cerrado formado por esta *tela*, le ponemos dentro una fuente de aire, que le ayuda a mantener su forma más o menos constante durante las deformaciones permitidas por los resortes que lo forman; ésa es la idea principal de nuestro modelo.

En la Figura 2.1 podemos ver un pequeño esquema del modelo, en este caso es un hexágono en dos dimensiones. Desde luego, mientras más compleja sea la forma de nuestro modelo, mas difícil será trabajar con él, pero se verá más realista

2.2.1 Esbozo general del algoritmo

Una vez conocida la idea, nos empezaremos a preguntar en cómo llevarlo a cabo, es decir cómo podemos implementar este modelo en forma de un algoritmo que seamos capaces de codificar en algún lenguaje de programación.

El diagrama de flujo del algoritmo es el que se muestra en la Figura 2.2, en donde se puede apreciar que sólo se tienen bloques de proceso; cada uno de estos procesos puede

¹Este modelo es muy socorrido cuando se trata de modelar ropa o tela, se pueden ver ejemplos en el último capítulo de [3], que modela una bandera o en [17] y [13], ambos modelan ropa sobre una persona

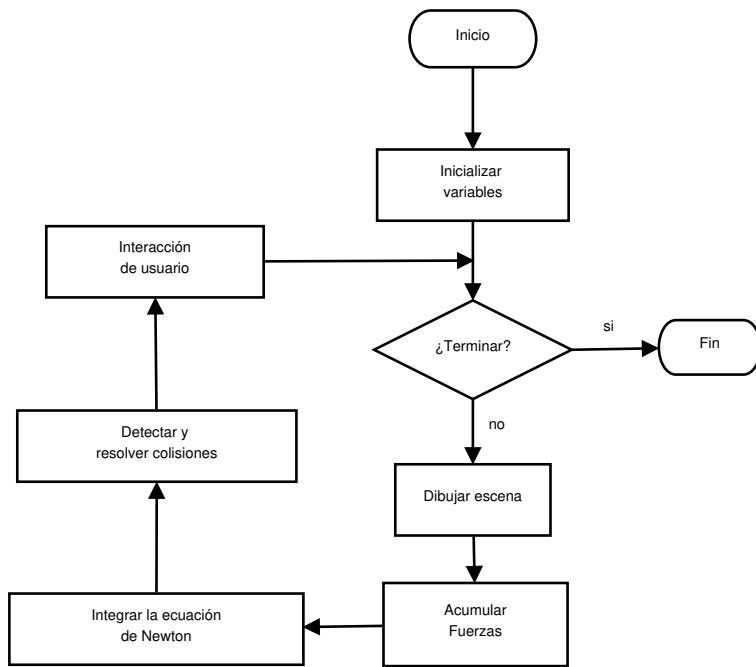


Figura 2.2: Este diagrama de flujo muestra, los detalles generales a seguir en una simulación gráfica basada en física. Para cada paso del algoritmo se pueden tener diferentes estrategias, pero en general este esquema no debe cambiar.

ser llevado a cabo de muchas maneras, pero tendrá que ser en este orden, en esta sección veremos con un poco más de detalle cada proceso.

2.2.1.1 Inicializar variables

Lo primero es dar valor a las variables que lo necesiten, me refiero con esto a variables globales, que afectan cómo funciona el programa por el resto de su ejecución. Dentro de estas variables están los parámetros externos del modelo, es decir esos que no pueden ser calculados dentro del programa y que es necesario que se proporcionen por el usuario. Los parámetros necesarios son m para cada partícula, g , k_d y k_s para los resortes, L para cada resorte en particular, y k_g para la acumulación de la fuerza debida a la presión.

Otra cosa que debe hacerse en este paso del algoritmo es poner las condiciones iniciales del modelo, por ejemplo, la posición de cada partícula y la velocidad al momento de iniciar la animación. Y de igual manera con los cuerpos no flexibles que deseemos que interactúen con nuestro modelo, se debe de conocer su posición, si se mueven o no, y cómo van a interactuar con el modelo.

Por último las opciones globales, que afecten a la animación también se deben de obtener aquí, por ejemplo el tamaño del paso Δt para mover el modelo, las opciones de los gráficos (si se hará *render*, o se tratará sólo con el *wireframe*²) y qué tipo de integrador se ocupará en el modelo. Alguien me podría decir que estos también son parámetros. Y en efecto lo son, quise sin embargo hacer la distinción y llamarlos variables globales, por el hecho de que no se deben al modelo en sí, sino más bien a la implementación que cada quien haga del mismo, es decir se deben al programa.

2.2.1.2 Dibujar escena

Esta parte del algoritmo debe ser la encargada de dibujar en la pantalla toda la escena. Para hacer esto debe contar con la información de donde se encuentran cada punto y donde se encuentra cada cosa que queramos dibujar.

Aunque no hay mucho que decir aquí, ésta parte es delicada, depende de nuestro conocimiento de las herramientas que vamos a usar para hacer el programa. Con la ayuda de una biblioteca gráfica, en este caso *OpenGL*, y los datos necesarios guardados de una manera ordenada, como en una estructura de datos, debemos de ser capaces de implementar esta función.

El detalle de la implementación depende del modelo en cuestión, veré a detalle la que yo use. Esto puede ser útil para que alguien que deseé implementar una simulación, se de algunas ideas, pero como ya dije, esta parte depende del fenómeno a modelar, por lo que en cada implementación se debe de hacer un análisis.

2.2.1.3 Acumular fuerzas

Las fuerzas que actúan sobre cada partícula del modelo, se pueden identificar como *externas*, es decir que se deben al medio, o *internas* que se deben al cuerpo en sí. Ejemplos de fuerzas externas son la gravedad, la viscosidad del medio, la resistencia del aire, por mencionar algunas. Ejemplos de las fuerzas internas son por otra parte: los *resortes amortiguadores* y la debida a la presión del gas.

²Cuando sólo se dibujan los bordes de las figuras, como si éstas estuvieran hechas de una malla de alambre se le llama *wireframe*, cuando dibujamos el área de las figuras y les damos color y efectos de iluminación se dice que se les dio *render*.

Cualquiera que sea el caso, en este paso debemos de calcular todas las fuerzas que intervienen en nuestro modelo, y debemos de acumularsela a cada uno de las partículas que lo conforman, es decir, cada partícula debe tener asociada la suma de todas las fuerzas que actúan sobre ella. Recordemos que la fuerza en general es un vector, por lo que la acumulación de la que hablo es una suma vectorial de cada una de las fuerzas.

Los detalles de cómo acomular las fuerzas específicas que ocuparé en la simulación se darán en la siguiente sección y con más a detalle en el resto del capítulo.

2.2.1.4 Integrar la ecuación de Newton

Las fuerzas acomuladas en el paso anterior, junto con la posición y velocidad actuales de la partícula, nos proporcionan (como dice la segunda ley de Newton (1.2)) el siguiente estado, nos dan la información suficiente para saber la nueva posición y la nueva velocidad de la partícula en cuestión. Conociendo esto para todas las partículas tendremos determinado el estado completo del modelo en el tiempo siguiente.

Como ésta es una ecuación diferencial, debemos de resolverla, o mejor dicho integrarla, para conocer el siguiente estado del sistema, sin embargo como nuestra acumulación de fuerzas fue hecha a partir de muchas fuerzas, todas ellas de naturaleza distinta, no podremos integrar esta ecuación de manera analítica. Por esta razón debemos de recurrir a un método numérico que nos permita aproximar la solución.

Por suerte hay muchísimos métodos numéricos que se ajustan a nuestras necesidades, aquí he decidido presentar a dos de ellos solamente, en la sección dedicada a integrar la ecuación de Newton explicaré a detalle cómo funcionan. De manera general, se puede decir que todos ellos toman como información la posición y velocidad actual de la partícula, el vector fuerza que actúa sobre ella y el instante de tiempo que pasa entre el estado actual y el nuevo estado que queremos calcular (ésta cantidad es Δt). Y después de hacer cálculos con ellos nos devuelven un nuevo estado, en forma de la nueva posición y la nueva velocidad de la partícula.

2.2.1.5 Detectar y resolver colisiones

En este paso debemos analizar qué pasa cuando nuestro modelo choca o interactúa con otros objetos de la escena. Estos otros objetos pueden ser cuerpos rígidos como el piso

o flexibles como otro cuerpo de la misma característica del nuestro.

El segundo paso es la respuesta a esta colisión. Este problema es también delicado, pero en general, se va a encargar de ver la partícula que chocó, moverla como respuesta al choque y modificar su estado, es decir su posición y su velocidad. En una sección posterior de este capítulo hablaré con más detalle y también daré algunas ideas generales.

2.2.1.6 Interacción del usuario

Este paso es opcional, y depende de si deseamos que la simulación tenga una forma de ejecutarse independiente del usuario o si se desea que éste pueda cambiar la simulación en tiempo de ejecución, por ejemplo modificando algún parámetro de la simulación.

Para enfrentar esto se tiene que recurrir totalmente a la creatividad del programador que vaya a hacer la implementación. Una buena idea es por ejemplo utilizar alguna biblioteca que exista precisamente para este fin. En éste caso se decidió utilizar un par de bibliotecas que se integran bien con *OpenGL*. Por un lado *glfw*, que permite manejar eventos del ratón y del teclado. Y por otro lado *Dear ImGui* que permite implementar una interfaz gráfica (menú) de usuario.

2.3 El sistema de fuerzas

Es momento de hacernos unas preguntas: ¿Qué es lo que compone nuestro cuerpo? ¿Qué necesitamos saber para implementar este modelo? La primera pregunta es sencilla: nuestro cuerpo está formado por puntos, que están unidos por resortes, que a su vez se unen para formar las caras. Resumiendo, nuestro cuerpo es un conjunto de caras, que a su vez son un conjunto de resortes, que a su vez son un conjunto de puntos: ¡Nuestro cuerpo está formado por puntos (partículas)! La respuesta de la segunda interrogante está ligada a la primera, necesitamos saber todo lo que sea necesario para poder acumular fuerza a las partículas, es decir que para cada una debemos de poder ocupar las ecuaciones (1.3), (1.22) y (1.24).

Ciertamente cada una de estas fuerzas es de naturaleza distinta, la fuerza de gravedad es externa y se puede calcular para cada partícula por separado, la de los resortes depende

Fuerza	Símbolo	Parámetros	Datos necesarios
Gravedad	\mathbf{F}_g	g, m	Ninguno
Resorte	$\mathbf{F}_s + \mathbf{F}_d$	k_s, k_d	Partículas unidas por el resorte Velocidad y posición de cada partícula que une el resorte
Presión	\mathbf{F}_p	k_g	Partículas que forman la cara Volumen total del cuerpo Área de la cara

Cuadro 2.1: Fuerzas a calcular para cada partícula y que se necesita saber para hacer los cálculos

de la posición de cada partícula y de sus vecinos a los cuales está conectado por un *resorte-amortiguador* y por último la de la presión depende de la cara en la que esté la partícula y del volumen total del cuerpo. Podemos resumirlo en el Cuadro 2.1.

Para calcular la fuerza de gravedad es necesario que se conozcan dos parámetros, la masa de cada partícula m , y la constante gravitación g . La gravedad debe de acumularse en cada partícula.

La fuerza del *resorte amortiguador* es una fuerza interna. Para calcularla necesitamos recibir dos parámetros: k_s y k_d . Luego necesitamos calcular la posición y la velocidad de cada partícula, y saber por cada resorte qué partículas está uniendo. Cuando tengamos que acumular esta fuerza recorreremos todos los resortes y por cada resorte calcularemos una fuerza, luego la acomularemos en cada uno de las dos partículas a los que este resorte esté conectado.

Por último, para la fuerza debida a la presión, necesitamos un parámetro externo: k_g . Y para poder calcular esta fuerza debemos conocer antes de empezar el volumen total del cuerpo V , luego debemos de saber qué partículas pertenecen a cada una de las caras del cuerpo. Para acumular esta fuerza debemos recorrer cada una de las caras que forman el cuerpo, en cada caso calculamos el área de la cara y acumulamos la fuerza \mathbf{F}_p que le corresponda en cada una de las partículas que la forman.

2.4 Área, volumen y vectores normales

Los parámetros necesarios para calcular la fuerza de presión son el volumen total del cuerpo, el área de cada una de las caras, y un vector normal a dicha cara; como ya se dijo, esto depende enteramente de la geometría del cuerpo que se quiera modelar. Sin embargo, esto no quiere decir que no se puedan dar algunas técnicas generales que en alguna medida puedan ser adaptadas a alguna geometría particular. El propósito de esta sección es precisamente el de explicar esas técnicas generales.

2.4.1 Cálculo de áreas

Para calcular el área de un cuerpo geométrico generalmente se calcula el área de cada una de sus caras y después se suman. Empezando por el caso más simple, supongamos que tenemos un triángulo formado por tres puntos, sabemos las coordenadas de cada uno de los puntos, y queremos su área. Supongamos que los puntos son denotados por **P**, **Q** y **R**. Podemos calcular dos de las aristas del triángulo si calculamos los vectores que van de **P** a **Q** y de **P** a **R**. Luego podríamos calcular el producto cruz de los dos vectores que acabamos de encontrar y obtener su norma, finalmente la mitad de la norma sería el área del triángulo que estamos buscando.

$$\overrightarrow{PQ} = \mathbf{Q} - \mathbf{P}$$

$$\overrightarrow{PR} = \mathbf{R} - \mathbf{P}$$

$$A = \frac{1}{2} |\overrightarrow{PQ} \times \overrightarrow{PR}|$$

Si queremos calcular el área de un cuadrilátero podemos usar la idea anterior y *triangular*, el cuadrilátero en dos partes, supongamos que queremos calcular el área del cuadrilátero formado por los puntos **P**, **Q**, **R** y **S**, podríamos calcular el área del triángulo $\triangle PQR$ y luego sumar el área del triángulo $\triangle SRQ$. Dicho de otra manera:

$$\begin{aligned}
 A_T &= A_{\triangle \mathbf{PQR}} + A_{\triangle \mathbf{SRQ}} \\
 A_T &= \frac{1}{2} |\overrightarrow{\mathbf{PQ}} \times \overrightarrow{\mathbf{PR}}| \times \frac{1}{2} |\overrightarrow{\mathbf{SQ}} \times \overrightarrow{\mathbf{SR}}| \\
 A_T &= \frac{1}{2} (|\overrightarrow{\mathbf{PQ}} \times \overrightarrow{\mathbf{PR}}| + |\overrightarrow{\mathbf{SQ}} \times \overrightarrow{\mathbf{SR}}|) \tag{2.1}
 \end{aligned}$$

Este resultado de hecho, puede ampliarse aún más con el enunciado siguiente:

Sea P un polígono simple, sin agujeros, dado por la secuencia ordenada de vértices $\mathbf{P}_i = (x_i, y_i)$, $i = 1, \dots, n$, (por ejemplo en el sentido contrario a las agujas del reloj), entonces el área de P es:

$$A(P) = \frac{1}{2} \sum_{i=1}^{n-1} \det(\mathbf{P}_i, \mathbf{P}_{i+1}) + \frac{1}{2} \det(\mathbf{P}_n, \mathbf{P}_1)$$

Este teorema fue tomado de [12] donde también es demostrado. Con la adecuada combinación de estas técnicas es sencillo ingeníárselas para poder calcular el área de una cara.

2.4.2 Cálculo de volúmenes

Calcular volúmenes es usualmente más complejo que calcular áreas, y por ende casi siempre es computacionalmente costoso, de ahí que además de maneras geométricas de calcular un volumen siempre está la alternativa de aproximar un volumen. De nuevo hay que pensar qué queremos lograr con la simulación si rapidez en la ejecución, o apego a la realidad física.

No hay que olvidar también que para todo modelo y sus respectivos parámetros, siempre hay diferentes grados de sensibilidad, por ejemplo en el caso de nuestro modelo el volumen sólo es utilizado para poder determinar la fuerza escalar de la presión, por lo que es un modelo *poco sensible* al cálculo del volumen. Es decir, es válido aproximarse en este rubro en este rubro, sin perder mucha realidad en el modelo.

2.4.2.1 Volúmenes aproximados

Hay varias formas de aproximar el volumen de un cuerpo, una de ellas es con cajas envolventes o *bounding boxes*, la idea es muy simple se trata de aproximar el volumen de un cuerpo complejo mediante el volumen de un cuerpo simple que lo contenga. Es decir, se aproxima el volumen por medio de poliedros regulares ya sean inscritos o circunscritos, tales que den una buena aproximación del cuerpo cuyo volúmen estemos calculando.

Una forma más simple es aproximando con un *hexaedro regular* cuyo volumen es $l \cdot w \cdot h$, o largo por alto por ancho; otra forma geométrica que se usa de manera muy común es el *elipsoide*, cuyo volumen es $\frac{4\pi}{3}a \cdot b \cdot c$, donde a , b y c , son los largos de sus ejes principales.

2.4.2.2 Volúmenes Exactos

Una técnica muy útil para calcular el volúmen de una malla triangular se puede ver en [19].

La técnica consiste recorrer todas las caras, y en cada una calcular el *volumen orientado* del tetraedro cuya base es la cara triangular y cuya punta es el origen del sistema de referencia. El signo del volumen (por eso dijimos que es un volumen orientado) depende de si el tetraedro apunta hacia el origen. Eso quiere decir que habrá algunos volúmenes que sean negativos y otros positivos, la suma de todos los volúmenes es el volumen de la malla.

Es importante que al usar esta fórmula, los triángulos apunten hacia el mismo lado de la malla. En otras palabras, se deben de definir un orden para los vértices de los triángulos (por ejemplo en contra de las manecillas del reloj) y se debe ser consistente en toda la malla. En efecto, esta técnica usa una normal a la cara del triángulo para orientar, pero dicha normal está implícita por el orden de los vértices.

En resumen el volumen de la malla formada por n caras triangulares está dado por:

$$V = \frac{1}{6} \sum_{i=1}^n \mathbf{g}_i \cdot \mathbf{N}_i \quad (2.2)$$

En donde \mathbf{g}_i y \mathbf{N}_i son el baricentro y la normal del triángulo i . Y asumiendo que dicho triángulo tiene vértices en el orden \mathbf{v}_i^1 , \mathbf{v}_i^2 y \mathbf{v}_i^3 se pueden calcular de la siguiente manera:

$$\mathbf{g}_i = \frac{(\mathbf{v}_i^1 + \mathbf{v}_i^2 + \mathbf{v}_i^3)}{3} \quad (2.3)$$

$$\mathbf{N}_i = (\mathbf{v}_i^2 - \mathbf{v}_i^1) \times (\mathbf{v}_i^3 - \mathbf{v}_i^1) \quad (2.4)$$

Como ya se dijo, se puede transformar cualquier malla poligonal en una malla triangular subdividiendo cada cara en triángulos. Por ejemplo, nuestro cuerpo neumático está formado por caras en forma de cuadriláteros. Cada cara se puede partir por su diagonal para formar dos triángulos.

2.4.3 Vectores normales

Dado un plano podemos encontrar un vector que le sea normal, sólo necesitamos dos vectores linealmente independientes que se encuentren en el plano y calculando su producto cruz obtenemos un vector normal, es decir en términos más simples, si conocemos tres puntos del plano y no son colineales, podemos calcular el vector perpendicular a ese plano.

Matemáticamente: dado el triángulo formado por los puntos $\mathbf{A}_1 = (x_1, y_1, z_1)$, $\mathbf{A}_2 = (x_2, y_2, z_2)$ y $\mathbf{A}_3 = (x_3, y_3, z_3)$, un vector normal a este plano, es el vector \mathbf{n} que se calcula:

$$\mathbf{n} = \overrightarrow{\mathbf{A}_1\mathbf{A}_2} \times \overrightarrow{\mathbf{A}_1\mathbf{A}_3}$$

Un hecho muy importante es que si queremos calcular el vector normal a una superficie, podemos calcular los vectores normales a cada vértice de ella y luego promediarlos. Esto por increíble que parezca funciona y es de hecho la manera como se hacen los cálculos de iluminación en la mayoría de los programas de CAD y de simulación gráfica.

Es decir, si tenemos el polígono de n lados, formado por los puntos $\mathbf{P}_1, \mathbf{P}_2, \dots, \mathbf{P}_n$, ordenados de alguna manera, por ejemplo en sentido contrario a las manecillas del reloj, y queremos el vector normal a este polígono \mathbf{n} , calculamos los vectores $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$,

de la forma $\mathbf{v}_i = \overrightarrow{\mathbf{P}_i \mathbf{P}_{i+1}} \times \overrightarrow{\mathbf{P}_i \mathbf{P}_{i+1}}$ para $i = 2, \dots, n$ y $\mathbf{v}_1 = \overrightarrow{\mathbf{P}_1 \mathbf{P}_2} \times \overrightarrow{\mathbf{P}_1 \mathbf{P}_n}$, y luego el vector normal \mathbf{n} es:

$$\mathbf{n} = \frac{1}{n} \sum_{i=1}^n \mathbf{v}_i \quad (2.5)$$

Este hecho es fundamental para nuestros cálculos, tanto de la fuerza debida a la presión, como para la iluminación en el render. Una demostración se puede encontrar en [12].

2.5 Integrar la ecuación de Newton

Desde que hay modelos físicos, ha habido interés en la solución numérica de ciertos problemas para los que se sabía de antemano que una solución existía, pero no se contaban con métodos analíticos para encontrarla. Es por esta razón que nace el análisis numérico, sin embargo no fue hasta que se popularizó el uso de las computadoras que este campo tomó más importancia.

Dentro del análisis numérico, es de nuestro interés la solución numérica de ecuaciones diferenciales. El problema comienza con una ecuación de primer orden con condición inicial y que cumple las condiciones de existencia y unicidad.

Actualmente, hay muchas maneras de atacar estos problemas, es decir muchas familias de métodos. Por familia quiero decir que cuando alguien propone un método y llega otra persona más y le hace correcciones, ahora hay dos métodos pero en esencia funcionan con la misma idea, por eso son dos métodos de la misma familia.

El primer método para obtener una aproximación numérica de la solución de una ecuación diferencial es el método de Euler, y desde entonces se han propuesto muchísimos métodos e ideas más. El Cuadro 2.2 es una línea del tiempo que tiene algunos de los acontecimientos más importantes del desarrollo de esta disciplina.

En esencia estos métodos se encargan de resolver la ecuación diferencial de la forma:

$$\begin{aligned} \frac{d\mathbf{y}}{dt} &= \mathbf{F}(t, \mathbf{y}) \\ \mathbf{y}(t_0) &= \mathbf{y}_0 \end{aligned} \quad (2.6)$$

Año	Evento
1768	Leonhard Euler publica su método, el primero en la historia
1824	Agustin Cauchy demuestra la convergencia del método de Euler, emplea el método de Euler implícito
1855	En una carta escrita por John F. Bashforth, se mencionan por primera vez los métodos de pasos múltiples de Couch Adams
1895	Carl Runge publica el primer método de Runge Kutta
1905	Martin Kutta describe el popular método de Runge Kutta de orden cuatro
1910	Lewis Fry Richardson anuncia su método de extrapolación.
1952	Charles F. Curtiss y Joseph Oakland Hirschfelder acuñan el término <i>stiff equations</i> .
1967	Loup Verlet publica su método, especialmente enfocado a la mecánica de partículas

Cuadro 2.2: Una línea de tiempo que muestra algunos de los acontecimientos más importantes para la solución numérica de ecuaciones diferenciales

Y para hacerlo toman la condición inicial (2.6), como el primer valor de la solución, con ella calculan un valor aproximado para la solución $\mathbf{y}(t)$ en el tiempo $t = t_0 + \Delta t$. Ahora este nuevo punto de la solución lo llamamos \mathbf{y}_n y nos ayuda a encontrar un nuevo punto dando otro paso hacia adelante en el tiempo $t = t_0 + 2\Delta t$. Y así sucesivamente, de manera que la salida de nuestro método son una colección de parejas $(t + n\Delta t, \mathbf{y}(t + n\Delta t))$ la primera de ellas es la condición inicial, y de ahí en adelante se trata de aproximaciones de la solución.

Sin embargo nosotros no queremos resolver una ecuación como ésta, deseamos resolver la ecuación (1.2), que es una ecuación diferencial de segundo orden. Para poder adaptar este problema al método se propone un cambio de variable $\frac{d\mathbf{x}}{dt} = \mathbf{v}(t)$, por lo que el problema se transforma en resolver un sistema de dos ecuaciones diferenciales.

$$\begin{aligned}\frac{d\mathbf{v}}{dt} &= \frac{1}{m} \mathbf{F}(\mathbf{x}, \mathbf{v}, t) \\ \frac{d\mathbf{x}}{dt} &= \mathbf{v}(t) \\ \mathbf{v}(0) &= \mathbf{v}_0 \\ \mathbf{x}(0) &= \mathbf{x}_0\end{aligned}$$

Los métodos siguientes entonces supondrán que sabemos la velocidad y la posición de la partícula en el tiempo t , así como la fuerza que actúa sobre ella en este tiempo, con esta última podremos calcular la posición y la velocidad de la partícula en un tiempo posterior $t + \Delta t$.

2.5.1 El método de Euler

Este método fue el primero de todos, de ahí que también sea el más simple, sin embargo aún tiene algunas ventajas el usarlo. Básicamente el método de Euler nos dice que si tenemos una ecuación de la forma:

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y})$$

Con una condición inicial de la forma $\mathbf{y}(0) = \mathbf{y}_0$.

Entonces podemos aproximar el siguiente punto de la solución con la siguiente fórmula de recurrencia:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{f}(t_n, \mathbf{y}_n)$$

En donde h representa el tamaño del paso en el tiempo hacia adelante es decir $h = \Delta t$.

El método de Euler se puede generalizar para sistemas de ecuaciones diferenciales de tamaño n . Estas generalizaciones se pueden ver en casi cualquier libro de Ecuaciones y en cualquier libro de análisis numérico. Aquí sólo daré las fórmulas de recurrencia para el caso de $n = 2$ por ser el que estamos interesados en resolver.

Dado el sistema:

$$\begin{aligned}\mathbf{x}'(t) &= \mathbf{f}(t, \mathbf{x}, \mathbf{y}) \\ \mathbf{y}'(t) &= \mathbf{g}(t, \mathbf{x}, \mathbf{y})\end{aligned}\tag{2.7}$$

Con las condiciones iniciales:

$$\begin{aligned}\mathbf{x}(t_0) &= \mathbf{x}_0 \\ \mathbf{y}(t_0) &= \mathbf{y}_0\end{aligned}\tag{2.8}$$

Entonces la solución se puede aproximar con las siguientes fórmulas de recurrencia:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\mathbf{f}(t_n, \mathbf{x}_n, \mathbf{y}_n)$$

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{g}(t_n, \mathbf{x}_n, \mathbf{y}_n)$$

Ahora vamos a poner nuestro sistema de manera que podamos ocupar estas fórmulas para resolverlo:

$$\begin{aligned}\mathbf{v}'(t) &= \frac{1}{m}\mathbf{F}(t, \mathbf{x}, \mathbf{v}) \\ \mathbf{x}'(t) &= \mathbf{v}(t, \mathbf{x}, \mathbf{v})\end{aligned}\tag{2.9}$$

Y nuestras condiciones iniciales son:

$$\begin{aligned}\mathbf{v}(t_0) &= \mathbf{v}_0 \\ \mathbf{x}(t_0) &= \mathbf{x}_0\end{aligned}\tag{2.10}$$

Por lo tanto para resolver nuestro sistema con el método de Euler se tiene:

$$\begin{aligned}\mathbf{v}_{n+1} &= \mathbf{v}_n + \frac{h}{m}\mathbf{F}(t_n, \mathbf{x}_n, \mathbf{v}_n) \\ \mathbf{x}_{n+1} &= \mathbf{x}_n + h\mathbf{v}_{n+1}\end{aligned}\tag{2.11}$$

2.5.1.1 El error del método de Euler

Como en todo procedimiento numérico, en el método de Euler se tiene un error, el cual puede encontrarse para nuestro caso por medio de la expansión en la serie de Taylor para la función que determina la posición de una partícula.

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\mathbf{v}_n$$

$$\mathbf{x}(t_0 + h) = \mathbf{x}(t_0) + h\mathbf{v}(t_0)$$

Pero sabemos que la serie de Taylor de la trayectoria de una partícula es:

$$\mathbf{x}(t_0 + h) = \mathbf{x}(t_0) + h\mathbf{x}'(t_0) + \frac{1}{2}h^2\mathbf{x}''(t_0) + \mathbf{O}(h^3)$$

Entonces el error en el método de Euler está dado por la diferencia de ambas expresiones es decir:

$$-\frac{1}{2}h^2\mathbf{x}''(t_0) + \mathbf{O}(h^3)$$

Este error es el error de truncamiento, o debido al método en sí. Al momento de implementarlo existe también un error por redondeo, que es debido a que las computadoras operan con un número finito de decimales, sin embargo este error es difícil de estimar y sale del alcance de mi investigación. De aquí en adelante cuando me refiera al error de un método numérico siempre me referiré al *error por truncamiento*.

2.5.1.2 Ventajas y desventajas del método de Euler

Como todo algoritmo este método presenta ventajas y desventajas, éstas se deben tanto al método como a su implementación en este modelo. Aquí listaré algunas de ellas.

Ventajas:

- Se puede implementar fácilmente.
- Se puede integrar partícula por partícula, dado que sólo requiere de una evaluación de \mathbf{F} .

- Es rápido de ejecutarse.

Desventajas:

- Tiende a *explotar* rápidamente.
- Su error es alto.
- Es muy sensible a las variaciones pequeñas, por lo que tarda en estabilizarse.

2.5.2 El método de Runge Kutta

Supongamos que se tiene una ecuación de la forma

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y})$$

Con su respectiva condición inicial de la forma $\mathbf{y}(0) = \mathbf{y}_0$.

El método de Runge y Kutta nos dice que la solución en el siguiente paso de tiempo puede aproximarse con la siguiente fórmula de recurrencia:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h(\mathbf{k}_n^1 + 2\mathbf{k}_n^2 + 2\mathbf{k}_n^3 + \mathbf{k}_n^4)}{6}$$

En donde h representa el tamaño del paso en el tiempo que se desee aproximar y los coeficientes $\mathbf{k}_n^1, \mathbf{k}_n^2, \mathbf{k}_n^3, \mathbf{k}_n^4$ se pueden calcular de la siguiente manera:

$$\begin{aligned}\mathbf{k}_n^1 &= \mathbf{f}(t_n, \mathbf{y}_n) \\ \mathbf{k}_n^2 &= \mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{h}{2}\mathbf{k}_n^1\right) \\ \mathbf{k}_n^3 &= \mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{h}{2}\mathbf{k}_n^2\right) \\ \mathbf{k}_n^4 &= \mathbf{f}(t_n + h, \mathbf{y}_n + h\mathbf{k}_n^3)\end{aligned}$$

Al igual que en el método de Euler, existe una generalización del método de Runge Kutta para poderse ocupar con sistemas de ecuaciones de primer orden. En casi cualquier libro

de ecuaciones se pueden ver estas fórmulas (por ejemplo en [2]); aquí solo pongo el caso $n = 2$ porque es el que voy a ocupar en el modelo.

Supongamos que tenemos de nuevo el sistema (2.7), sujeto a (2.8). Podemos aproximar una solución usando el método de Runge Kutta con la siguiente fórmula de recurrencia, suponiendo que queremos ir de t_n a t_{n+1} .

$$\begin{aligned}\mathbf{x}_{n+1} &= \mathbf{x}_n + \frac{h(\mathbf{k}_n^1 + 2\mathbf{k}_n^2 + 2\mathbf{k}_n^3 + \mathbf{k}_n^4)}{6} \\ \mathbf{y}_{n+1} &= \mathbf{y}_n + \frac{h(\mathbf{l}_n^1 + 2\mathbf{l}_n^2 + 2\mathbf{l}_n^3 + \mathbf{l}_n^4)}{6}\end{aligned}$$

Aquí podemos apreciar que ahora tenemos que encontrar ocho ponderadores, los \mathbf{k}_i^j y los \mathbf{l}_i^j significa que tenemos más cálculos por hacer. Los valores de los ponderadores se calculan con las siguientes fórmulas.

$$\begin{aligned}\mathbf{k}_n^1 &= \mathbf{f}(t_n, \mathbf{x}_n, \mathbf{y}_n) \\ \mathbf{k}_n^2 &= \mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{x}_n + \frac{h}{2}\mathbf{k}_n^1, \mathbf{y}_n + \frac{h}{2}\mathbf{l}_n^1\right) \\ \mathbf{k}_n^3 &= \mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{x}_n + \frac{h}{2}\mathbf{k}_n^2, \mathbf{y}_n + \frac{h}{2}\mathbf{l}_n^2\right) \\ \mathbf{k}_n^4 &= \mathbf{f}(t_n + h, \mathbf{x}_n + h\mathbf{k}_n^3, \mathbf{y}_n + h\mathbf{l}_n^3)\end{aligned}$$

y

$$\begin{aligned}\mathbf{l}_n^1 &= \mathbf{g}(t_n, \mathbf{x}_n, \mathbf{y}_n) \\ \mathbf{l}_n^2 &= \mathbf{g}\left(t_n + \frac{h}{2}, \mathbf{x}_n + \frac{h}{2}\mathbf{k}_n^1, \mathbf{y}_n + \frac{h}{2}\mathbf{l}_n^1\right) \\ \mathbf{l}_n^3 &= \mathbf{g}\left(t_n + \frac{h}{2}, \mathbf{x}_n + \frac{h}{2}\mathbf{k}_n^2, \mathbf{y}_n + \frac{h}{2}\mathbf{l}_n^2\right) \\ \mathbf{l}_n^4 &= \mathbf{g}(t_n + h, \mathbf{x}_n + h\mathbf{k}_n^3, \mathbf{y}_n + h\mathbf{l}_n^3)\end{aligned}$$

El método de Runge Kutta, es mucho más exacto que el método de Euler, de hecho

se puede apreciar que el método de Euler, aproxima el siguiente paso con el valor de una pendiente, en cambio el método de Runge Kutta aproxima el siguiente paso con el valor de cuatro pendientes ponderadas, cada una calculada con la aproximación de la anterior.

Supongamos que se tiene de nuevo (2.9) sujeto a las condiciones (2.10), entonces el método de Runge Kutta toma la siguiente forma:

$$\begin{aligned}\mathbf{v}_{n+1} &= \mathbf{v}_n + \frac{h}{6}(\mathbf{k}_n^1 + 2\mathbf{k}_n^2 + 2\mathbf{k}_n^3 + \mathbf{k}_n^4) \\ \mathbf{x}_{n+1} &= \mathbf{x}_n + \frac{h}{6}(l_n^1 + 2l_n^2 + 2l_n^3 + l_n^4)\end{aligned}\quad (2.12)$$

Y los ponderadores se pueden calcular de la siguiente manera:

$$\begin{aligned}\mathbf{k}_n^1 &= \frac{1}{m}\mathbf{F}(t_n, \mathbf{x}_n, \mathbf{v}_n) \\ l_n^1 &= \mathbf{v}_n \\ \mathbf{k}_n^2 &= \frac{1}{m}\mathbf{F}\left(t_n + \frac{h}{2}, \mathbf{x}_n + \frac{h}{2}\mathbf{k}_n^1, \mathbf{v}_n + \frac{h}{2}l_n^1\right) \\ l_n^2 &= \mathbf{v}_n + \frac{h}{2}\mathbf{k}_n^1 \\ \mathbf{k}_n^3 &= \frac{1}{m}\mathbf{F}\left(t_n + \frac{h}{2}, \mathbf{x}_n + \frac{h}{2}\mathbf{k}_n^2, \mathbf{v}_n + \frac{h}{2}l_n^2\right) \\ l_n^3 &= \mathbf{v}_n + \frac{h}{2}\mathbf{k}_n^2 \\ \mathbf{k}_n^4 &= \frac{1}{m}\mathbf{F}(t_n + h, \mathbf{x}_n + h\mathbf{k}_n^3, \mathbf{v}_n + hl_n^3) \\ l_n^4 &= \mathbf{v}_n + h\mathbf{k}_n^3\end{aligned}\quad (2.13)$$

2.5.2.1 El error del método de Runge Kutta

Se puede demostrar por medios algebraicos y de la misma manera que se empleó con el método de Euler que el método de Runge Kutta tiene un error del orden de $\mathbf{O}(h^5)$. Es decir si partimos el intervalo h por la mitad, y damos el doble de pasos el error por truncamiento disminuye en un orden de 16 veces (es decir $(\frac{h}{2})^4$, como el error es del

orden de $O(h^5)$ se hace h^4 veces más exacto).

2.5.2.2 Ventajas y desventajas del método de Runge Kutta

Como se puede leer en la bibliografía, el método de Euler y el método de Runge Kutta dependen del tamaño del paso h , y se hacen más exactos mientras el paso es más pequeño, pero también se hacen más cálculos, por lo que aumenta la fuente del otro tipo de error, el error por redondeo. Se acepta de manera general, que de este tipo de métodos, aquel que optimiza esta situación es el método de Runge Kutta de cuarto orden, el método que acabamos de presentar.

Algunas de las ventajas y desventajas para el método de Runge Kutta son las siguientes.

Ventajas

- Es el método estándar más recomendado y no es tan difícil de implementar (métodos más exactos son considerablemente más difíciles de implementar).
- Es recomendado por muchos autores y por lo tanto hay mucha documentación.
- Es rápido de ejecutarse, (Más lento que el Euler, pero para efectos de la animación la diferencia no es apreciable).
- Es un método muy estable, es muy difícil que explote.

Desventajas

- A diferencia de Euler, requiere de muchas evaluaciones de la función en diferentes puntos (lo que para nosotros significa calcular muchas veces la fuerza sobre la partícula).
- Las partículas no se pueden integrar por separado, tendrán que integrarse juntas. (todas las que conforman el cuerpo en un solo paso).

2.6 Cómo enfrentar la colisión de los cuerpos

Por colisión de los cuerpos entenderemos la manera cómo lidiar cuando dos cuerpos dentro de la escena quedan en contacto uno con el otro. Desde nuestro punto de vista

esta respuesta se divide en dos pasos: la detección de la colisión y la respuesta a la colisión.

La detección es totalmente un problema geométrico, consiste en que a partir de la información que tenemos de los cuerpos y de la colisión podemos determinar el punto de la colisión y un vector normal a ese punto, este problema es totalmente dependiente de la forma de los cuerpos. Por otro lado, la respuesta a la colisión es un problema físico y generalmente es más sencillo debido a que ya existen algoritmos bien definidos para responder a las colisiones y son independientes de la geometría, es decir son algoritmos generales.

En nuestro caso el cuerpo flexible está formado por partículas. Para resolver las colisiones consideramos cada partícula como un cuerpo independiente y tratamos la posible colisión de cada una con la esfera por separado.

2.6.1 La detección de las colisiones

Como ya se dijo antes la detección de las colisiones es uno de los problemas más complejos que nos podemos enfrentar al hacer una animación. Básicamente debemos de poder hacer dos cosas aquí: uno, detectar la colisión, es decir mediante una prueba rápida saber si los cuerpos de nuestra escena entraron en contacto, y después ver si podemos determinar un vector normal al punto (o superficie en 3D) de colisión.

Hay básicamente dos tipos de estrategias para resolver este problema, una es por medio de un *bounding volume*, que se trata de darle la vuelta al problema con una aproximación y otra es por medios estrictamente geométricos es decir tratar de dividir tus cuerpos en figuras de formas elementales, como esferas o paralelepípedos para los cuales la detección es un poco más sencilla.

2.6.1.1 Un bounding volume

Básicamente se trata de imaginar que hay un envolvente de nuestro objeto y este envolvente tiene una forma más sencilla, entonces al probar por la colisión se prueba con el envolvente no con el objeto. Esto tiene la enorme desventaja de ser una aproximación, por lo que la animación se verá un poco saltada, sin embargo una cuidadosa elección del objeto envolvente hará que este efecto sea mínimo.

Objetos como las elipses, los paralelepípedos y los cilindros son buenos objetos para ser un *bounding volume*, porque la detección de la colisión es sencilla.

Por ejemplo, en una esfera podemos determinar si un punto esta dentro de ella con tan sólo comparar el cuadrado de su distancia con respecto al centro de la esfera, con el cuadrado del radio.

Un cilindro con el eje vertical alineado con el eje de la escena es también muy usado. Si queremos saber si dos objetos contenidos en un cilindro, por ejemplo en un videojuego dos personajes caminando, chocan, sólo debemos ver si la proyección de los ejes principales de los cilindros sobre el eje de la escena (líneas rectas) se interceptan y además las proyecciones de los dos cilindros con el plano *XZ* (dos círculos) también se intersecan.

Un paralelepípedo o rectángulo, o *bounding box*, también se usa mucho, por ejemplo cuando la geometría de los objetos hace a la esfera una mala elección, por ejemplo al ver si dos coches chocan en un juego de carreras el rectángulo es una elección mas sabia.

Un rectángulo usado como *bounding box* generalmente se alinea con las coordenadas del mundo, para así hacer las pruebas de detección triviales, si por el contrario el rectángulo es alineado con las coordenadas del objeto y este tiene la capacidad de rotar, cada vez que lo haga se necesita volver a calcular el *bounding box*.

2.6.1.2 Uniendo diferentes geometrías

Una manera más exacta de predecir si dos objetos de una escena están en colisión, es descomponer la forma completa de un objeto en varios objetos pequeños, y luego probar contra todos los objetos que componen el cuerpo si es que existió la colisión. Por ejemplo, en el caso de la simulación del cuerpo flexible se podría pensar como que cada partícula que forma el cuerpo es un objeto y luego probar la colisión contra todas las partículas que forman el cuerpo flexible.

Esta técnica es bastante más cara, tanto de implementar como de ejecutarse, sin embargo, da resultados visiblemente más acertados, por ejemplo en un juego de lucha libre, donde la interacción de los luchadores debe de ser bastante creíble, se recomendaría usar este tipo de colisiones.

2.6.2 La respuesta de las colisiones

Como ya habíamos dicho, el trabajo difícil y dependiente de la animación está en la detección de las colisiones, y la respuesta a las colisiones es totalmente física. Para empezar las colisiones se dividen en dos: elásticas e inelásticas. La implementación de ambas no se diferencia mucho, aunque sí son diferentes en concepto.

Básicamente en un algoritmo de respuesta de colisiones, se suministran las posiciones y las velocidades de los dos objetos que colisionan, más un vector normal unitario al punto de colisión de los dos objetos, o plano normal en el caso de tres dimensiones. Desde luego que hay dos vectores que cumplen con esa condición, cualquiera de ellos nos servirá siempre y cuando sepamos cuál de ellos es el que tenemos. Ya con esta información debemos ser capaces de responder con dos cosas: una nueva posición de los objetos y una nueva velocidad.

En la Figura 2.3 se ven los diferentes pasos de la respuesta a la colisión de dos círculos.

2.6.2.1 Separar un vector en componentes normal y tangencial

Antes de explicar la forma en que se responden las colisiones quiero hacer énfasis en la manera cómo separar un vector en componentes ortogonales (Figura 2.4), porque es necesario hacerlo en la respuesta a las colisiones.

Para separar a un vector cualquiera \mathbf{v} en dos componentes: uno normal \mathbf{v}_n , y otro tangencial \mathbf{v}_t respecto a un vector normal \mathbf{n} , se hace uso de las siguientes fórmulas:

$$\begin{aligned}\mathbf{v}_n &= \frac{(\mathbf{v} \cdot \mathbf{n})}{|\mathbf{n}|} \frac{\mathbf{n}}{|\mathbf{n}|} \\ \mathbf{v}_t &= \mathbf{v} - \mathbf{v}_n\end{aligned}$$

Como en los cálculos de detección de colisiones es común que tengamos un vector normal $\vec{\mathbf{n}}$, tal que: $|\vec{\mathbf{n}}| = 1$, las fórmulas anteriores se simplifican aún más siendo la forma más usada las ecuaciones (2.14)³:

³Las fórmulas (2.14), se encuentran mal escritas en [1], y esta fue una de las razones que más me retrasó al momento de hacer este trabajo.

Objeto	Información conocida	Información por determinar
A	Masa m , Velocidad \mathbf{u}	Velocidad ajustada \mathbf{s}
B	Masa p , Velocidad \mathbf{v}	Velocidad ajustada \mathbf{w}

Cuadro 2.3: Condiciones de la respuesta a las colisiones

$$\begin{aligned}\mathbf{v}_n &= (\mathbf{v} \cdot \vec{\mathbf{n}}) \vec{\mathbf{n}} \\ \mathbf{v}_t &= \mathbf{v} - \mathbf{v}_n\end{aligned}\tag{2.14}$$

2.6.2.2 Colisiones elásticas

Una colisión elástica es aquella donde el momento y la energía de los objetos, se conservan después de la colisión; son una abstracción que nunca sucede en la vida real. Aun las colisiones en el espacio exterior son inelásticas aunque están muy cerca de no serlo [9], pero nos sirven bastante para entender el fenómeno, y en algunos casos son suficientes. Por ejemplo, pensemos que estamos modelando un juego de billar en 2D una colisión elástica es suficiente.

Supongamos que se tienen dos cuerpos A y B (Figura 2.3a), las formas no importan, y sabemos, gracias a un algoritmo de detección de colisiones, que están en colisión (Figura 2.3b) y un vector normal $\vec{\mathbf{n}}$ a la superficie de colisión. Suponemos que este vector apunta del cuerpo A al cuerpo B y que es también un vector unitario (Figura 2.3c).

Suponemos también que conocemos todas las propiedades de los cuerpos, es decir su masa, su velocidad y su posición.

Queremos determinar una nueva posición y una nueva velocidad para los objetos como resultado de la colisión entre ellos (Figura 2.3d). Esto se resume en el Cuadro 2.3.

Para entender el porqué de la respuesta a la colisión, es necesario seguir los siguientes pasos⁴. Primero vamos por el caso más simple: dos objetos A y B , con velocidades \mathbf{u} y \mathbf{v} respectivamente chocan, como respuesta a la colisión las velocidades de ambos objetos cambian a \mathbf{s} y \mathbf{w} . Ver la Figura 2.3.

⁴Esta es una reproducción del procedimiento mostrado en [9] con mi nomenclatura

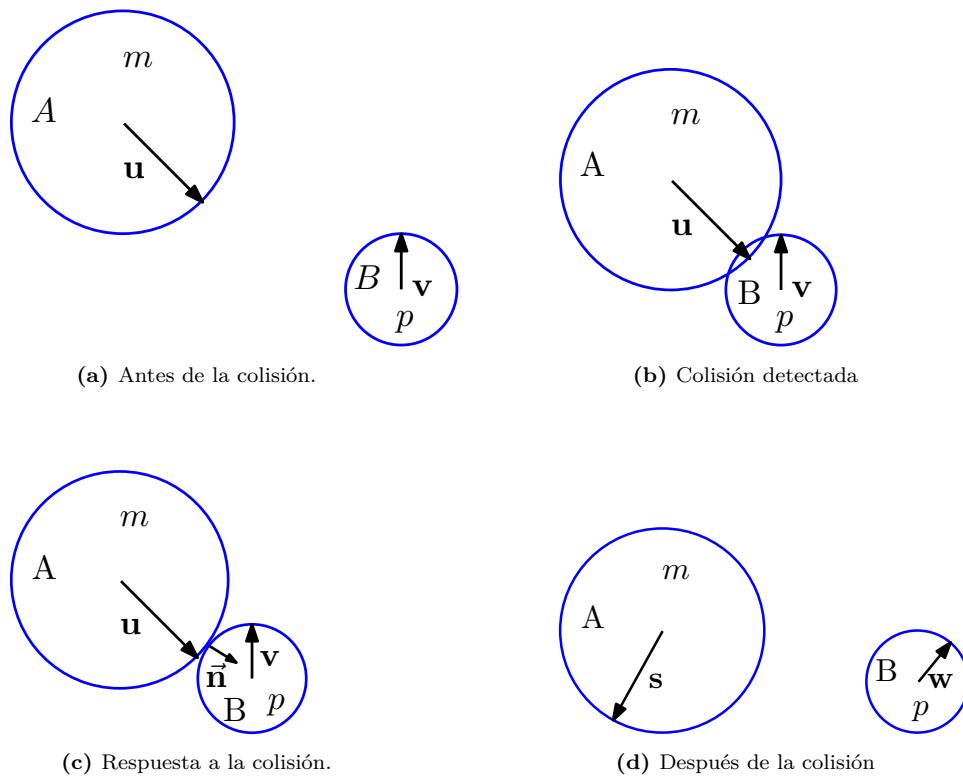


Figura 2.3: Etapas de detección y respuesta a una colisión

Suponemos que: $\mathbf{v} = \mathbf{0}$. Es decir el cuerpo B no se mueve, está detenido esperando la colisión del cuerpo A . Sabemos que tenemos un vector \vec{n} normal al plano de colisión. Con este vector podemos dividir los demás vectores en una parte normal al plano y otra tangencial (Figura 2.4), es decir:

$$\begin{aligned}\mathbf{u} &= \mathbf{u}_t + \mathbf{u}_n \\ \mathbf{v} &= \mathbf{v}_t + \mathbf{v}_n \\ \mathbf{s} &= \mathbf{s}_t + \mathbf{s}_n \\ \mathbf{w} &= \mathbf{w}_t + \mathbf{w}_n\end{aligned}$$

Ahora veamos qué es lo qué sabemos de antemano por la forma como planteamos las condiciones del problema: sabemos que: $\mathbf{u}_t = \mathbf{s}_t$ y $\mathbf{v}_t = \mathbf{w}_t = \mathbf{0}$, porque esperaríamos que las velocidades sólo se vieran afectadas en su componente normal y porque sabemos que el objeto B estaba inicialmente en reposo. Así que lo único que necesitamos saber

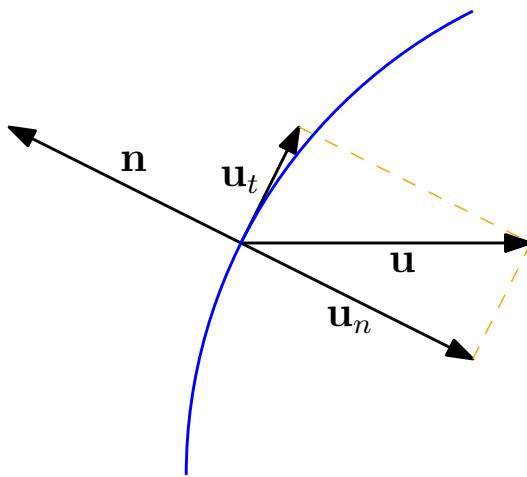


Figura 2.4: El vector \mathbf{u} se separa con respecto a \mathbf{n} en dos vectores uno normal \mathbf{u}_n , y uno tangencial \mathbf{u}_t

es \mathbf{s}_n y \mathbf{w}_n .

Se sabe también que, como la colisión es elástica, debe de obedecer la ley de la conservación de la energía (2.16) y la la ley de la conservación de momento (2.15).

$$m\mathbf{u}_n = m\mathbf{s}_n + p\mathbf{w}_n \quad (2.15)$$

$$\frac{1}{2}m\mathbf{u}^2 = \frac{1}{2}m\mathbf{s}^2 + \frac{1}{2}p\mathbf{w}^2 \quad (2.16)$$

Así que tenemos justo las condiciones necesarias para encontrar una solución, pues tenemos dos valores por determinar y dos ecuaciones que las relacionan. Para encontrar la solución hacemos un cambio de variable $r = \frac{m}{p}$. Podemos encontrar una expresión para \mathbf{w}_n de la ecuación (2.15), y lo sustituimos en (2.16) para obtener el valor de \mathbf{s}_n . En realidad se obtienen dos valores $\mathbf{s}_n^1 = \mathbf{u}_n$ y $\mathbf{s}_n^2 = \mathbf{u}_n \frac{r-1}{r+1}$, tomamos el segundo (el primero corresponde a la condición inicial, por que las colisiones elásticas son reversibles en el tiempo), y lo sustituimos de nuevo en (2.15) para obtener el valor de $\mathbf{w}_n = \frac{2r\mathbf{u}_n}{r+1}$.

Con esto se tiene todo lo necesario para resolver una colisión elástica con uno de los dos objetos en reposo. Podemos utilizar el siguiente pseudocódigo:

Algoritmo 1 Respuesta a una colisión elástica

```

1: procedure RESPUESTACOLISIONELASTICA(u, n, m, p)
2:    $r \leftarrow \frac{m}{p}$ 
3:   un  $\leftarrow$  PARTENORMAL(u, n)
4:   ut  $\leftarrow$  u - un
5:   sn  $\leftarrow$  un  $\left(\frac{r-1}{r+1}\right)$ 
6:   wn  $\leftarrow$  un  $\left(\frac{2r}{r+1}\right)$ 
7:   s  $\leftarrow$  ut + sn
8:   w  $\leftarrow$  wn
9:   return s, w
10: end procedure
11: procedure PARTENORMAL(v, n)
12:   vn  $\leftarrow \frac{(\mathbf{v} \cdot \mathbf{n})}{|\mathbf{n}|} \frac{\mathbf{n}}{|\mathbf{n}|}$ 
13:   return vn
14: end procedure

```

En donde la función `parteNormal`, es una función que recibe un vector **u** y un vector **n**, devuelve la parte normal de **u** con respecto a **n**. Es decir, nos sirve para partir al vector **u** en una parte tangencial y una parte normal a la colisión haciendo uso de las fórmulas (2.14).

Ahora veamos el caso más general, donde **v** $\neq \mathbf{0}$. Aquí vamos a ocupar el principio de relatividad y le restamos a todo el sistema **v**, lo que lo transforma en el caso anterior. Resolvemos como lo habíamos hecho antes y luego le sumamos a todo el sistema **v**. El pseudocódigo es casi idéntico:

Algoritmo 2 Respuesta a una colisión elástica con ambos cuerpos en movimiento

```

1: procedure RESPUESTACOLISIONELASTICA(u, v, n, m, p)
2:    $r \leftarrow \frac{m}{p}$ 
3:   u  $\leftarrow \mathbf{u} - \mathbf{v}$ 
4:   un  $\leftarrow \text{PARTENORMAL}(\mathbf{u}, \mathbf{n})$ 
5:   ut  $\leftarrow \mathbf{u} - \mathbf{u}_{\mathbf{n}}$ 
6:   sn  $\leftarrow \mathbf{u}_{\mathbf{n}} \left( \frac{r-1}{r+1} \right)$ 
7:   wn  $\leftarrow \mathbf{u}_{\mathbf{n}} \left( \frac{2r}{r+1} \right)$ 
8:   s  $\leftarrow \mathbf{u}_t + \mathbf{s}_n + \mathbf{v}$ 
9:   w  $\leftarrow \mathbf{w}_n + \mathbf{v}$ 
10:  return s, w
11: end procedure

```

Con esto podemos programar un función general que resuelva colisiones elásticas, sólo debemos asegurarnos de que la función que detecta las colisiones le informe a aquella función de tres cosas: las propiedades del objeto *A*, las propiedades del objeto *B* y un vector normal al plano de colisión que vaya de *A* a *B*.

Vuelvo a hacer énfasis en que para nuestro caso, la colisión de cada partícula con la esfera debe resolverse de manera separada e independiente.

2.6.2.3 Colisiones inelásticas

En una colisión inelástica se tiene una pérdida de energía como respuesta al impacto. En la realidad las colisiones son inelásticas, podemos apreciar parte de la pérdida de la energía, al escuchar el sonido de la colisión. Por ejemplo en el billar.

Para simular una colisión inelástica, vamos también a hacer una suposición: que los objetos tienen una cierta eficiencia, y que ésta se mantiene fija para todas las colisiones que involucran ese objeto. Es una gran simplificación porque en la realidad la eficiencia de una colisión depende de muchos factores, como el medio ambiente, o la velocidad de los objetos al momento de la colisión.

Si dos objetos chocan y uno de ellos tiene *coeficiente de restitución* de 0.9 y el otro un coeficiente de restitución de 0.85, la energía total después de la colisión sería: $(0.9 \cdot 0.85) (E_b + E_a)$. Donde E_a y E_b es la energía de cada objeto antes de la colisión.

Como se puede ver, el coeficiente de restitución es una manera de medir la *eficiencia* y significa que el primer objeto después de una colisión transmite por ejemplo el 95 por ciento de su energía.

Para resolver una colisión inelástica se usa el mismo procedimiento que en la sección anterior, sólo que ahora la ecuación (2.16), toma la siguiente forma:

$$\frac{1}{2}em\mathbf{u}^2 = \frac{1}{2}m\mathbf{s}^2 + \frac{1}{2}p\mathbf{w}^2 \quad (2.17)$$

Donde e es el producto de los coeficientes de restitución de ambos objetos. Y se procede de la misma manera que en el caso anterior a calcular los valores de \mathbf{w}_n y \mathbf{s}_n . Resolviendo el sistema formado por las ecuaciones (2.15) y (2.17).

Las nuevas soluciones son:

$$\begin{aligned}\mathbf{s}_n &= \frac{-r\mathbf{u} - \sqrt{r^2\mathbf{u}_n^2 - (r+1)((r-e)\mathbf{u}_n^2 + (1-e)\mathbf{u}_t^2)}}{r+1} \\ \mathbf{w}_n &= r(n\mathbf{u}_n - \mathbf{v}_n)\end{aligned}$$

Y el pseudocódigo, que resuelve la colisión de manera inelástica es el siguiente:

Algoritmo 3 Respuesta general a una colisión inelástica

```

1: procedure RESPUESTACOLISIONINELASTICA( $\mathbf{u}, \mathbf{v}, \mathbf{n}, m, p, E_a, E_b$ )
2:    $r \leftarrow \frac{m}{p}$ 
3:    $e \leftarrow E_a \cdot E_b$ 
4:    $\mathbf{u} \leftarrow \mathbf{u} - \mathbf{v}$ 
5:    $\mathbf{u}_n \leftarrow \text{PARTEMNORMAL}(\mathbf{u}, \mathbf{n})$ 
6:    $\mathbf{u}_t \leftarrow \mathbf{u} - \mathbf{u}_n$ 
7:    $d \leftarrow r^2 (\mathbf{u}_n \cdot \mathbf{u}_n) - ((r-e)(\mathbf{u}_n \cdot \mathbf{u}_n) + (1-e)(\mathbf{u}_t \cdot \mathbf{u}_t))$ 
8:    $\mathbf{s}_n \leftarrow \mathbf{n} \left( \frac{\sqrt{d}-r\mathbf{u}_n}{r+1} \right)$ 
9:    $\mathbf{w}_n \leftarrow r \left( \frac{\mathbf{u}_n-\mathbf{v}_n}{r+1} \right)$ 
10:   $\mathbf{s} \leftarrow \mathbf{u}_t + \mathbf{s}_n + \mathbf{v}$ 
11:   $\mathbf{w} \leftarrow \mathbf{w}_n + \mathbf{v}$ 
12:  return  $\mathbf{s}, \mathbf{w}$ 
13: end procedure
```

Capítulo 3

Implementación del modelo

Este capítulo está dedicado a cubrir los detalles de la implementación del modelo en código, concretamente en lenguaje C++.

Dado que la principal finalidad de este trabajo es el modelado físico, no entrará en detalles en otras áreas del programa más que las que tienen que ver con lo descrito en los capítulos anteriores.

El programa completo contiene módulos para la graficación en [OpenGL](#), para la interfaz de usuario por medio de [dear imgui](#), para la lectura y escritura de imágenes usando [freeimage](#) (para cargar texturas y para hacer capturas de pantalla) e implementa el modelo de iluminación de [Phong](#) en shaders por medio de [GLSL](#).

El lector interesado puede consultar el código fuente completo del programa que se incluye en un CD o puede descargarlo de [github](#). Aunque el código está escrito en inglés (por ser el estándar) traté de ser lo más extenso en los comentarios en todo el código. Por lo que el lector que cuente con los conocimientos de programación y graficación por computadora adquiridos en la licenciatura podrá entender y usar todo el código fuente del programa sin ningún problema.

Se decidió implementar el programa usando el paradigma de programación orientada a objetos. En la primera sección se describen las clases más importantes: las necesarias para representar las estructuras del modelo y la clase que representa el cuerpo neumático.

La siguiente sección se revisa cómo se implementaron en código los métodos numéricos

usados para integrar la ecuación de Newton.

En la tercera sección se explican las rutinas de la física del modelo, concretamente las rutinas que se encargan de aplicar las fuerzas que intervienen en el modelo.

En la última sección del capítulo, se explica cómo es que se implementaron la rutina tanto de detección como de respuesta a las colisiones.

3.1 Diseño de clases

“Show me your flowchart and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won’t usually need your flowchart; it’ll be obvious.”

Frederick P. Brooks
Mythical Man-Month

Decidí usar de la biblioteca de álgebra para gráficos `glm` por dos razones: Primero, porque cuenta con tipos de datos para representar vectores en 2 y 3 dimensiones e implementa las operaciones usuales entre ellos. Y segundo, porque esta biblioteca tiene un alto nivel de integración con OpenGL al grado de ser casi un estándar.

La parte fundamental del modelo la constituyen las partículas; de cada una de éstas nos interesa saber básicamente cuatro cosas: su posición, su velocidad, la fuerza que se le está aplicando y la masa. Adicionalmente, por un detalle particular a nuestra implementación, se necesita saber si una partícula está fija. Si una partícula está fija, entonces se deben de ignorar las operaciones de integración. La clase `Particle` representa (abstrae) una partícula. Los métodos nos permiten editar y consultar estos cinco miembros. En este sentido la clase `Particle` es muy simple ¹.

Para representar a los resortes-amortiguadores se creó la clase `SpringDamper` que tiene como miembros *referencias* a las dos partículas que une éste resorte. Hago énfasis en que usaremos referencias, pues las partículas van ser usadas para formar varios objetos y se requiere que no sea necesario actualizar en cada objeto por separado cada que una

¹En el argot del lenguaje `Java` diríamos que la clase `Particle` es casi un `bean`

partícula sea modificada. Finalmente, se debe saber la longitud L del resorte cuando está en su estado de reposo.

La clase `SpringDamper` define los métodos para:

- Actualizar qué partículas forman éste resorte.
- Consultar y modificar L .
- Consultar las posiciones de las partículas que forman éste resorte.
- Acumular la fuerza del resorte en sus dos partículas dados k_s y k_d .

La última clase necesaria para definir el cuerpo neumático es la clase `Face`, que representa una cara del cuerpo flexible. Como es de esperar, dado que las caras son cuadriláteros, esta clase solo necesita como propiedades referencias a cuatro partículas.

Adicionalmente, la clase `Face` implementa los siguientes métodos:

- Los accesores y mutadores a las partículas que la forman.
- Un método para actualizar las partículas que forman esta cara (equivalente al de `SpringDamper`).
- Un método para calcular el área de esta cara.
- Un método que sirve como auxiliar en el cálculo del volumen de todo el cuerpo flexible. La lógica de este método se explica en la Sección 3.3.3.²

La clase que representa el cuerpo flexible: `SoftBody` está compuesta con objetos de las clases anteriores. Esta clase es la más importante de la aplicación. Y el resto de éste capítulo está dedicado a explicar como funcionan sus métodos. En este momento, es relevante decir que esta clase contiene una colección de partículas, una colección de resortes y una colección de caras. Las caras y resortes en realidad usan (agregan) referencias a las partículas, de esta manera la información de las partículas sólo está almacenada una vez.

²Aunque ésta no es estrictamente hablando una propiedad de la cara, por razones de Ingeniería de Software (IS) era conveniente incluirlo en esta clase.

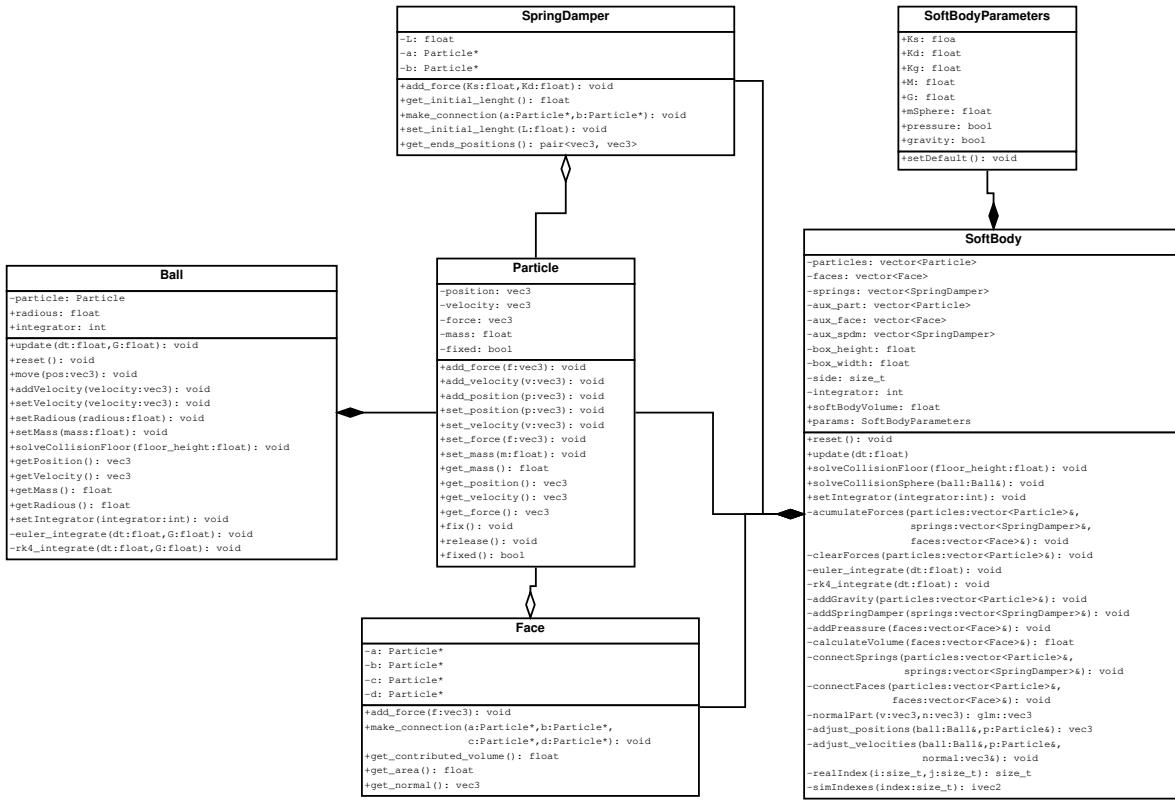


Figura 3.1: La clase **Particle** es la unidad de construcción. La clase **SoftBody** es el objeto principal que contiene a los demás.

Por orden, esta clase también define una estructura: **SoftBodyParameters** que contiene todos los parámetros físicos del cuerpo flexible.

Finalmente, se necesita también una clase para representar el único cuerpo rígido en la escena. La clase **Ball** tiene como miembros una partícula, un escalar que representa el radio de la esfera y un miembro para registrar qué método numérico se usa para integrar.³

La relación entre estas clases está resumida en el diagrama de clases mostrado en la Figura 3.1. De nuevo hago énfasis en que la implementación de contiene de hecho más miembros que los mostrados en dicha figura, pues solo aparecen en el diagrama los relevantes para éste trabajo.

³De nuevo por IS, era conveniente que cada objeto en la escena llevará registro de que integrador está en uso

3.2 Creación del cuerpo flexible

Cómo se dijo en la sección anterior, hay tres colecciones que forman el cuerpo flexible: las partículas, los resortes y las caras. Estas colecciones están almacenadas en arreglos dinámicos⁴ que forman parte de la clase `SoftBody`.

Hay algunas cosas que decir aquí: primero, que lo que se quiere modelar es una tela cuadrada que servirá como cuerpo flexible, por lo que el número de partículas es en realidad n^2 en donde n es el número de puntos en cada lado de la tela (en el código, n es el miembro `m_side` de `SoftBody`), y como no hay resortes en las orillas de la tela el número de resortes totales es $2(n - 1)(n - 2)$ y el número de caras es: $(n - 1)^2$. Segundo, que pese a que geométricamente la tela es un arreglo cuadrado de puntos, las partículas son guardados en un arreglo unidimensional (lineal), dado que esto simplifica y hace mucho más generales las rutinas de acumulación de fuerza.

Dado que la partículas están en un arreglo unidimensional se hace uso del método `simIndexes` que recibe el índice de la partícula en el vector unidimensional y nos regresa los índices (i, j) que la partícula tendría en un arreglo bidimensional. El método `realIndex` hace la operación inversa.

El código mostrado en el Listado 3.1 inicializa la velocidad, la fuerza y la masa de la partículas (la posición es inicializada con una rutina muy simple que está fuera de la clase `SoftBody`). También se encarga de fijar las partículas que están en las orillas de la tela.

La rutina que inicializa los resortes es ejecutada después que la de las partículas. Se muestra un extracto en el Listado 3.2.

Por las condiciones que definimos en nuestra tela.

- Las partículas que están en las aristas superiores e inferiores de la tela no tienen un resorte horizontal entre ellas.
- Las partículas que están en las aristas izquierda y derecha de la tela no tienen un resorte vertical entre ellas.

El algoritmo, pregunta si esta partícula puede ser unida (por un resorte) con la partícula

⁴En lenguaje C++ representados por objetos de tipo `std::vector`, los llamo *colecciones* en el texto para no confundirlos con los vectores de álgebra lineal

```

void SoftBody::recreate_particles(const mesh::Mesh& fabric) {
    // Destroy previous fabric
    m_particles.clear();
    m_aux_part.clear();

    // Get mesh vertices, since we will need the initial positions
    std::vector<mesh::Vertex> vertices = fabric.getVertices();

    assert(vertices.size() == (m_side * m_side));
    // Each particle's mass is total fabric mass divided by particles number
    const float P_MASS = 10.0f / (m_side * m_side);

    // Create particles
    for (size_t k = 0; k < vertices.size(); ++k) {
        const glm::vec3 velocity = glm::vec3(0.0);
        const glm::vec3 force = glm::vec3(0.0);
        physics::Particle p {vertices[k].position, velocity, force, P_MASS};
        // If you are on the edges get fixed
        glm::ivec2 indexes = simIndexes(k);
        size_t i = indexes.x;
        size_t j = indexes.y;
        if (i == (m_side - 1) || i == 0 || j == (m_side - 1) || j == 0) {
            p.fix();
        }
        m_particles.push_back(p);
        //Also make a copy to the aux
        m_aux_part.push_back(p);
    }
    // Create SpringDampers
    connectSprings(m_particles, m_springs);
    // copy for the aux
    connectSprings(m_aux_part, m_aux_spdm);
    //Create faces
    connectFaces(m_particles, m_faces);
    // copy for the aux
    connectFaces(m_aux_part, m_aux_face);

    mSoftBodyVolume = calculateVolume(m_faces);
}

```

Listado 3.1: El método `recreate_particles` de `SoftBody`

```
void SoftBody::connectSprings(std::vector<Particle>& particles, std::vector<SpringDamper>& springs) {
    // Empty the given array of spring - dampers
    springs.clear();
    m_spring_indices.clear();
    // Reconnect it using the particles array
    for (size_t k = 0; k < particles.size(); ++k) {
        // Get your relative indices
        glm::ivec2 indexes = simIndexes(k);
        size_t i = indexes.x;
        size_t j = indexes.y;
        // If you have a right neighbor and you are not in the upper or lower edge
        if (i < (m_side - 1) && j > 0 && j < (m_side - 1)) {
            SpringDamper sd{&particles[realIndex(i, j)], &particles[realIndex(i + 1, j)]};
            springs.push_back(sd);
            // Keep a reference of the connected indices (for wireframe rendering)
            m_spring_indices.push_back(realIndex(i, j));
            m_spring_indices.push_back(realIndex(i + 1, j));
        }
        // If you have down neighbor and you are not in the right or left edge
        if (j < (m_side - 1) && i > 0 && i < (m_side - 1)) {
            SpringDamper sd{&particles[realIndex(i, j)], &particles[realIndex(i, j + 1)]};
            springs.push_back(sd);
            // Keep a reference of the connected indices (for wireframe rendering)
            m_spring_indices.push_back(realIndex(i, j));
            m_spring_indices.push_back(realIndex(i, j + 1));
        }
    }
    assert(springs.size() == ((m_side - 1) * (m_side - 2) * 2));
}
```

Listado 3.2: El método connectSprings de SoftBody

```

void SoftBody::connectFaces(std::vector<Particle>& particles, std::vector<Face>& faces) {
    // Empty the given array of quad faces
    faces.clear();
    // Reconnect it using the given particles array
    for (size_t k = 0; k < particles.size(); ++k) {
        // Get your relative indices
        glm::ivec2 indexes = simIndexes(k);
        size_t i = indexes.x;
        size_t j = indexes.y;
        // If you have a both a right and a down neighbors
        if (i < (m_side - 1) && j < (m_side - 1)) {
            Face face{
                &particles[realIndex(i, j)], &particles[realIndex(i + 1, j)],
                &particles[realIndex(i, j + 1)], &particles[realIndex(i + 1, j + 1)]
            };
            faces.push_back(face);
        }
    }
    assert(faces.size() == (m_side - 1) * (m_side - 1));
}

```

Listado 3.3: El método connectFaces de SoftBody

a su derecha, de ser así las conecta. Después, pregunta si debe ser unida con la partícula debajo de ella, de ser así también es conectada.

La lógica para formar las caras se presenta en el Listado 3.3. Esta rutina es muy parecida a la anterior: recorre todas las partículas, si la partícula actual, tiene un partícula a su izquierda y otra partícula abajo, entonces se pueden tomar cuatro partículas (pues debe existir otra una partícula en diagonal) para formar una cara.

3.3 La física del modelo

Toca el turno de ver las rutinas que tienen que ver con la acumulación de fuerzas en el modelo. Como se ha dicho antes hay básicamente tres fuerzas que se deben acumular, la de la gravedad, la de los resortes amortiguadores y la debida a la presión del gas.

Aquí es donde empezaremos a notar el porqué de la construcción de los demás vectores de referencias.

```
void SoftBody::addGravity(std::vector<Particle>& particles) {
    const glm::vec3 UP = glm::vec3(0.0f, 1.0f, 0.0f);
    mTimers.gravityTime.tick();
    for (auto& particle : particles) {
        // Negative sign comes in the G value
        particle.add_force((particle.get_mass() * mParams.G) * UP);
    }
    mTimers.gravityTime.tock();
    mTimers.plotData[0] += mTimers.gravityTime.getDeltaMiliS();
}
```

Listado 3.4: El método `addGravity` de `SoftBody`

3.3.1 La fuerza de gravedad

La primera y más sencilla de las fuerzas que vamos a poner es la de gravedad. Como se dijo desde la ecuación (1.3), sólo depende de dos cosas de la masa del objeto y de la constante de gravedad⁵, y además sólo afecta el componente vertical (paralelo al eje *y*), del vector fuerza.

La fuerza de gravedad se aplica a cada una de las partículas del cuerpo flexible. Convenientemente, tenemos un vector con dichas partículas. Por lo anterior y gracias a que `glm` sobrecarga los operadores en vectores de la manera natural, la implementación es trivial como se puede ver en el Listado 3.4. La constante de gravedad es un miembro de `SoftBody` y tiene el signo negativo.

Para hacer posible el análisis de la Sección 4.3, se hacen uso de cronómetros. La estructura `mTimers` contiene estos cronómetros y el Listado 3.4 es un claro ejemplo de su uso. Estos cronómetros son una forma muy simplista de *instrumentación* de un método.

3.3.2 La fuerza de los resortes

La implementación de estos métodos también es trivial gracias a nuestro diseño. Tan solo se debe iterar por todos los resortes de `SoftBody` en cada resorte se debe ocupar la ecuación (1.22), para acumular la fuerza en los dos puntos que son unidos por ese resorte (Listado 3.5). Solo por completez, en Listado 3.6 se muestra el método `add_force` de `SpringDamper` que en esencia implementa la ecuación (1.22).

⁵La contribución de la masa será *cancelada* por el integrador

```
void SoftBody::addSpringDamper(std::vector<SpringDamper>& springs) {
    mTimers.springTime.tick();
    for (auto& spring : springs) {
        spring.add_force(mParams.Ks, mParams.Kd);
    }
    mTimers.springTime.tock();
    mTimers.plotData[1] += mTimers.springTime.getDeltaMiliS();
}
```

Listado 3.5: El método addSpringDamper de SoftBody

```
void SpringDamper::add_force(const float& Ks, const float& Kd) {
    glm::vec3 f, r, v;
    float f_s;

    r = (m_a->get_position() - m_b->get_position());
    v = (m_a->get_velocity() - m_b->get_velocity());

    f_s = Ks * (glm::length(r) - m_L) + Kd * (glm::dot(v, r) / glm::length(r));
    r = glm::normalize(r);
    f = f_s * r;

    m_b->add_force(f);
    m_a->add_force(-1.0f * f);
}
```

Listado 3.6: El método add_force de SpringDamper

```

void SoftBody::addPreassure(std::vector<Face>& faces) {
    mTimers.pressureTime.tick();
    // Update current volume
    mTimers.volumeTime.tick();
    mSoftBodyVolume = calculateVolume(faces);
    mTimers.volumeTime.tock();
    mTimers.plotData[2] += mTimers.volumeTime.getDeltaMiliS();
    for (auto& face : faces) {
        // Accumulate the pressure forces in the fabric faces
        glm::vec3 force = ((face.get_area() * mParams.Kg) / mSoftBodyVolume) * face.get_normal();
        face.add_force(force);
    }
    mTimers.pressureTime.tock();
    mTimers.plotData[3] += mTimers.pressureTime.getDeltaMiliS();
}

```

Listado 3.7: El método addPreassure de SoftBody

3.3.3 La fuerza del gas

La siguiente fuerza en ser acumulada es la fuerza debida a la presión del gas. Para esto se ocupa la ecuación (1.24), y se debe de acumular en cada cara. Para poder calcular esta fuerza se necesitan hacer varias operaciones importantes: calcular el volumen total del cuerpo flexible y además calcular el área y el vector normal de cada una de las caras (Listado 3.7).

Para hacer el cálculo de volumen se hace uso de la ecuación (2.2). Hay que recordar que el cuerpo neumático está formado por seis caras. Cinco de las cuales están fijas y otra es la tela (o tapa) que es el cuerpo flexible. Cada cara es dividida en dos triángulos para calcular su contribución al volumen. El Listado 3.8 muestra la lógica. La función `tethVolume` implementa las ecuaciones (2.3) y (2.4) y por completez se muestra en el Listado 3.9.

El método `calculateVolume` de la clase `Face`, suma la contribución de una cara al volumen usando la misma función `tethVolume` dos veces en los puntos que forman la cara (Listado 3.10)

Para calcular el área de cada cara se ocupa la fórmula (2.1), como se puede ver en el Listado 3.11. Y para calcular el vector normal se hace uso de la fórmula (2.5), en donde $n = 4$, dado que cada cara está formada por cuatro puntos (Listado 3.12).

```

float SoftBody::calculateVolume(std::vector<Face>& faces) {
    using namespace math;

    float volume{0.0f};
    // First, the fixed faces.
    // Start by calculating the corners of the box
    const glm::vec3 v0 = glm::vec3(-m_box_width / 2.0f, m_box_height, -m_box_width / 2.0f);
    const glm::vec3 v1 = glm::vec3( m_box_width / 2.0f, m_box_height, -m_box_width / 2.0f);
    const glm::vec3 v2 = glm::vec3(-m_box_width / 2.0f, m_box_height, m_box_width / 2.0f);
    const glm::vec3 v3 = glm::vec3( m_box_width / 2.0f, m_box_height, m_box_width / 2.0f);
    const glm::vec3 v4 = glm::vec3(-m_box_width / 2.0f, 0.0f, -m_box_width / 2.0f);
    const glm::vec3 v5 = glm::vec3( m_box_width / 2.0f, 0.0f, -m_box_width / 2.0f);
    const glm::vec3 v6 = glm::vec3(-m_box_width / 2.0f, 0.0f, m_box_width / 2.0f);
    const glm::vec3 v7 = glm::vec3( m_box_width / 2.0f, 0.0f, m_box_width / 2.0f);
    // Front face
    volume += tethVolume(v2, v7, v3);
    volume += tethVolume(v2, v6, v7);
    // Back face
    volume += tethVolume(v0, v1, v5);
    volume += tethVolume(v0, v5, v4);
    // Left face
    volume += tethVolume(v2, v0, v4);
    volume += tethVolume(v2, v4, v6);
    // Right face
    volume += tethVolume(v3, v5, v1);
    volume += tethVolume(v3, v7, v5);
    // Down face
    volume += tethVolume(v4, v5, v7);
    volume += tethVolume(v4, v7, v6);

    for (const auto& face : faces) {
        volume += face.get_contributed_volume();
    }

    return (1.0f / 6.0f) * volume;
}

```

Listado 3.8: El método calculateVolume de SoftBody

```

float tethVolume(const glm::vec3& v1, const glm::vec3& v2, const glm::vec3& v3) {

    glm::vec3 g_k = (1.0f / 3.0f) * (v1 + v2 + v3);
    glm::vec3 N_k = glm::cross(v2 - v1, v3 - v1);

    return glm::dot(g_k, N_k);
}

```

Listado 3.9: La función tethVolume implementa parte de la ecuación (2.2)

```

float Face::get_contributed_volume() const {
    using namespace math;
    float volume{0.0f};

    volume += tethVolume(m_a->get_position(), m_d->get_position(), m_b->get_position());
    volume += tethVolume(m_a->get_position(), m_c->get_position(), m_d->get_position());

    return volume;
}

```

Listado 3.10: El método get_contributed_volume de Face

```

float Face::get_area() const {
    //For the triangle ACB
    glm::vec3 cross_product = glm::cross(m_c->get_position() - m_a->get_position(), m_b->get_position() -
        ↪ m_a->get_position());
    float area = 0.5f * glm::length(cross_product);
    //For the triangle BCD
    cross_product = glm::cross(m_c->get_position() - m_d->get_position(), m_b->get_position() -
        ↪ m_d->get_position());
    area += 0.5f * glm::length(cross_product);

    return area;
}

```

Listado 3.11: El método get_area de Face

```

glm::vec3 Face::get_normal() const {

    const glm::vec3 atA = glm::normalize(glm::cross(m_c->get_position() - m_a->get_position(),
        ↪ m_b->get_position() - m_a->get_position()));
    const glm::vec3 atB = glm::normalize(glm::cross(m_a->get_position() - m_b->get_position(),
        ↪ m_d->get_position() - m_b->get_position()));
    const glm::vec3 atC = glm::normalize(glm::cross(m_d->get_position() - m_c->get_position(),
        ↪ m_a->get_position() - m_c->get_position()));
    const glm::vec3 atD = glm::normalize(glm::cross(m_b->get_position() - m_d->get_position(),
        ↪ m_c->get_position() - m_d->get_position()));

    return glm::normalize(0.25f * (atA + atB + atC + atD));
}

```

Listado 3.12: El método get_normal de Face

```
void SoftBody::euler_integrate(float dt) {
    const glm::vec3 ZERO = glm::vec3(0.0f);
    for (auto& particle : m_particles) {
        if (!particle.fixed()) {
            particle.add_velocity((dt / particle.get_mass()) * particle.get_force());
            particle.add_position(dt * particle.get_velocity());
        } else {
            particle.set_force(ZERO);
            particle.set_velocity(ZERO);
        }
    }
}
```

Listado 3.13: El método `euler_integrate` de `SoftBody`

3.4 Los métodos numéricos

Otras de las rutinas complicadas son los métodos numéricos. Para integrar la ecuación de Newton, se requiere saber la posición y la velocidad actual de cada una de las partículas del modelo y tener una manera de llamar a la función que se encarga de acumular las fuerzas.

3.4.1 El método de Euler

Básicamente se trata de tomar las ecuaciones (2.11) e implementarlas en el código (Listado 3.13), aprovechando la gran ventaja de que el método de Euler puede integrar una partícula la vez.

3.4.2 El método de Runge-Kutta

Aquí se explica el que fue probablemente el método numérico más complicado de implementar, pues no tiene la ventaja de Euler de integrar cada partícula independiente-mente, así que necesita integrar todas juntas. Además, debe tener espacio para guardar los ponderadores del paso de integración de cada partícula.

Para tener una idea clara de lo que hace el código en el Listado 3.14 conviene volver a ver las ecuaciones (2.13) y (2.12). Lo largo del proceso hacen ver el Listado 3.14 un poco más complicado que lo que en realidad es.

```
void SoftBody::rk4_integrate(float dt) {
```

```

// quick check for sync
assert(m_particles.size() == m_aux_part.size());
assert(m_springs.size() == m_aux_spdm.size());
assert(m_faces.size() == m_aux_face.size());
// create the 8 sets of ponderators
std::vector<glm::vec3> K1{m_particles.size(), glm::vec3(0.0f)};
std::vector<glm::vec3> K2{m_particles.size(), glm::vec3(0.0f)};
std::vector<glm::vec3> K3{m_particles.size(), glm::vec3(0.0f)};
std::vector<glm::vec3> K4{m_particles.size(), glm::vec3(0.0f)};
std::vector<glm::vec3> L1{m_particles.size(), glm::vec3(0.0f)};
std::vector<glm::vec3> L2{m_particles.size(), glm::vec3(0.0f)};
std::vector<glm::vec3> L3{m_particles.size(), glm::vec3(0.0f)};
std::vector<glm::vec3> L4{m_particles.size(), glm::vec3(0.0f)};

// Calculate first set of ponderators
for(size_t i = 0; i < m_particles.size(); ++i) {
    K1[i] = m_particles[i].get_force() / m_particles[i].get_mass();
    L1[i] = m_particles[i].get_velocity();
}
// Copy over the auxiliary
for(size_t i = 0; i < m_particles.size(); ++i) {
    m_aux_part[i].set_position(m_particles[i].get_position());
    m_aux_part[i].set_velocity(m_particles[i].get_velocity());
}
// Accumulate forces for the aux array
clearForces(m_aux_part);
acumulateForces(m_aux_part, m_aux_spdm, m_aux_face);
// Calculate second set of ponderators
for(size_t i = 0; i < m_aux_part.size(); ++i) {
    K2[i] = m_aux_part[i].get_force() / m_aux_part[i].get_mass();
    L2[i] = m_aux_part[i].get_velocity() + (dt / 2.0f) * K1[i];
}
// Copy over the auxiliary
for(size_t i = 0; i < m_aux_part.size(); ++i) {
    m_aux_part[i].set_position(m_particles[i].get_position() + (dt / 2.0f) * L2[i]);
    m_aux_part[i].set_velocity(m_particles[i].get_velocity() + (dt / 2.0f) * K2[i]);
}
// Accumulate forces for the aux array
clearForces(m_aux_part);
acumulateForces(m_aux_part, m_aux_spdm, m_aux_face);
// Calculate third set of ponderators
for(size_t i = 0; i < m_aux_part.size(); ++i) {
    K3[i] = m_aux_part[i].get_force() / m_aux_part[i].get_mass();
    L3[i] = m_aux_part[i].get_velocity() + (dt / 2.0f) * K2[i];
}
// Copy over the auxiliary
for(size_t i = 0; i < m_aux_part.size(); ++i) {
    m_aux_part[i].set_position(m_particles[i].get_position() + (dt / 2.0f) * L3[i]);
    m_aux_part[i].set_velocity(m_particles[i].get_velocity() + (dt / 2.0f) * K3[i]);
}
// Accumulate forces for the aux array
clearForces(m_aux_part);
acumulateForces(m_aux_part, m_aux_spdm, m_aux_face);
// Calculate fourth set of ponderators
for(size_t i = 0; i < m_aux_part.size(); ++i) {
    K4[i] = m_aux_part[i].get_force() / m_aux_part[i].get_mass();
    L4[i] = m_aux_part[i].get_velocity() + (dt / 2.0f) * K3[i];
}
// Now, that we have the ponderators, lets get deltas
std::vector<glm::vec3> deltaPos{m_particles.size(), glm::vec3(0.0f)};
std::vector<glm::vec3> deltaVel{m_particles.size(), glm::vec3(0.0f)};
for (size_t i = 0; i < m_particles.size(); ++i) {
    deltaPos[i] = (dt / 6.0f) * (L1[i] + (2.0f * (L2[i] + L3[i])) + L4[i]);
    deltaVel[i] = (dt / 6.0f) * (K1[i] + (2.0f * (K2[i] + K3[i])) + K4[i]);
}

```

```
// Finally, for each particle update position and velocity
for (size_t i = 0; i < m_particles.size(); ++i) {
    if (!m_particles[i].fixed()) {
        m_particles[i].add_position(deltaPos[i]);
        m_particles[i].add_velocity(deltaVel[i]);
    } else {
        const glm::vec3 ZERO = glm::vec3(0.0f);
        m_particles[i].set_force(ZERO);
        m_particles[i].set_velocity(ZERO);
    }
}
```

Listado 3.14 El método `rk4_integrate` de `SoftBody`

El truco consiste en ocupar un conjunto de puntos, resortes y caras auxiliares, para con ellos evaluar la función fuerza que es necesaria para calcular los ponderadores del método de RK4 (Por eso están presentes en el diagrama de la Figura 3.1), después la rutina es muy parecida a la de Euler.

3.5 El manejo de las colisiones

Como ya se ha dicho antes, el problema de las colisiones se resuelve en dos partes: primero la detección y luego la respuesta. La forma de responder consiste básicamente en mover los objetos que se colisionan a un lugar donde ya no choquen y ajustar las velocidades como respuesta.

3.5.1 La rutina de las colisiones

La detección es llevada a cabo en dos funciones. Recordemos que nuestra tarea de detección se simplifica muchísimo por el hecho de que uno de los objetos, el objeto incidente es una esfera. Para saber si dicha esfera está en colisión con nuestro cuerpo flexible, lo que hacemos es probar si cualquiera de las partículas está dentro de la esfera; de ser así, empezamos a resolver la colisión entre la esfera y la partícula en cuestión. Después seguimos revisando el resto de las partículas. El Listado 3.15 implementa el algoritmo.

```

void SoftBody::solveCollisionSphere(physics::Ball& ball) {
    const float radious_squared = ball.getRadius() * ball.getRadius();
    // For all particles
    for (auto& p : m_particles) {
        // If you are inside the sphere
        if (glm::distance2(ball.getPosition(), p.getPosition()) < radious_squared) {
            // Solving collision is done in two parts:
            // First, you move the object away of each other
            const glm::vec3 normal = adjust_positions(ball, p);
            // Second, you adjust the corresponding velocities
            adjust_velocities(ball, p, normal);
        }
    }
}

```

Listado 3.15: El método solveCollisionSphere de SoftBody

```

glm::vec3 SoftBody::adjust_positions(physics::Ball& ball, physics::Particle& p) {
    // Calculate collision's normal
    glm::vec3 normal = glm::normalize(p.getPosition() - ball.getPosition());
    // How to separate depends on the particle being fix
    if (p.fixed()) {
        // Move ball outside the particle along the normal
        ball.move(p.getPosition() - (ball.getRadius() * normal));
    } else {
        // Move particle outside the ball along the normal
        p.setPosition(ball.getPosition() + (ball.getRadius() * normal));
    }

    return normal;
}

```

Listado 3.16: El método adjust_positions de SoftBody

3.5.2 La detección de la colisión

El método del Listado 3.16 se encarga de calcular el vector normal al lugar de la colisión y de mover el punto fuera de la esfera. Mover el punto fuera de la esfera es en realidad parte de la respuesta a la colisión, pero decidí implementarlo en un solo método llamado `adjust_positions`, para que en la siguiente parte sólo se tenga que ajustar las velocidades.

El Listado 3.16, primero calcula el vector normal que va del centro de la esfera a la partícula. Luego, si la partícula no está fija, la mueve justo fuera de la esfera. Si la partícula estuviera fija, lo que hace es mover a la esfera fuera de ella.

```

void SoftBody::adjust_velocities(physics::Ball& ball, physics::Particle& p, const glm::vec3& normal) {
    // Use momentum equations to solve the elastic collision
    const float ratio = ball.getMass() / p.get_mass();
    // Calculate resultant velocity
    glm::vec3 u = ball.getVelocity() - p.get_velocity();
    // Split it in two components: normal and tangential with the respect of the collision plane
    glm::vec3 u_n = normalPart(u, normal);
    glm::vec3 u_t = u - u_n;
    // Calculate normal velocities after the collision
    glm::vec3 s_n = ((ratio - 1.0f) / (ratio + 1.0f)) * u_n;
    glm::vec3 w_n = ((2.0f * ratio) / (ratio + 1.0f)) * u_n;
    // Adjust ball's velocity
    ball.setVelocity(u_t + s_n + p.get_velocity());
    // Accumulate particle's return velocity
    p.add_velocity(w_n);
}

```

Listado 3.17: El método `adjust_velocities` de `SoftBody`

3.5.3 La respuesta de la colisión

La mayor parte del trabajo se hizo en la función anterior, ahora que sabemos el vector normal unitario: \vec{n} a la colisión, el Listado 3.17 solo se encarga de seguir los pasos explicados en el Algoritmo 2.

Es importante que la velocidad y posición de la esfera sean ajustadas de manera independiente por cada partícula con la que se haya producido una colisión. Es *físicamente incorrecto* tratar de acumular las posiciones y velocidades de respuesta de colisión con cada partícula y luego actualizar el cuerpo rígido una sola vez con las resultantes.

Capítulo 4

Diseño del experimento

En este capítulo se analizan los resultados que se obtuvieron al haber implementado el modelo, y se divide en tres partes. En la primera de ellas se cuenta cómo es que se decidió implementar el modelo, así como también cuales de sus variables se dejaron fijas (serían constantes en esta implementación) y cuáles es posible variar desde la interfaz de usuario.

En la segunda parte se diseñaron ciertas pruebas con el fin de poner en evidencia características particulares del modelo, específicamente su correcto comportamiento físico.

Y en la última parte se reportan los resultados de las pruebas de desempeño a las que se sometió el programa.

4.1 Definición del sistema

En esta primera parte presento el experimento y explico que parámetros fijé y cuáles pueden ser cambiados por el usuario.

4.1.1 Características del modelo

Tal como se explicó en la Sección 2.1, se modela un hexaedro regular, donde cinco de sus seis caras son rígidas y la cara superior o tapa es flexible: además, asumo que este

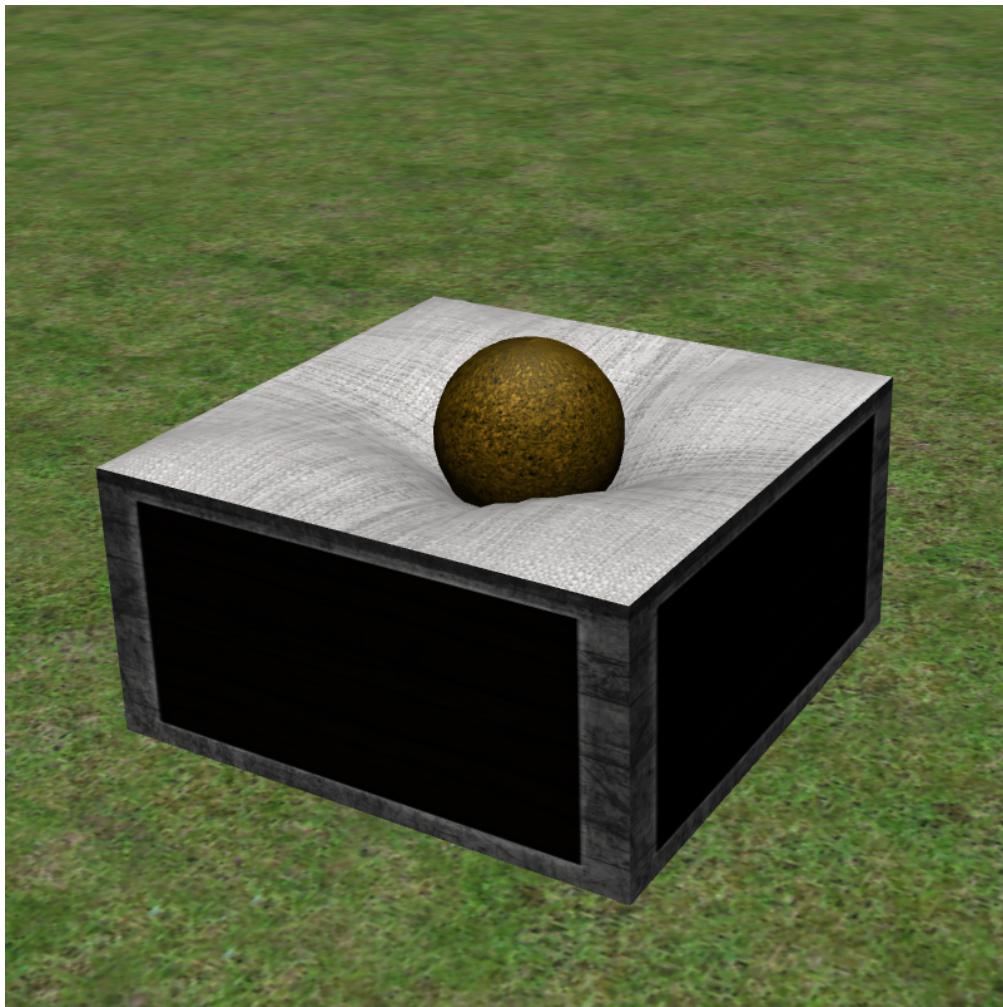


Figura 4.1: Una imagen del programa que implementa el modelo.

hexaedro está lleno de gas. En este sentido, el hexaedro completo (o caja) es el cuerpo neumático. Mientras que la cara superior (o tela) es el cuerpo flexible. Se deja caer sobre la tela una esfera rígida (o pelota). Un ejemplo del programa terminado se muestra en la Figura 4.1.

4.1.1.1 Constantes del experimento

En una situación como la antes descrita hay muchísimas variables del modelo, sin embargo al momento de hacer la implementación en código decidí dejar fijas algunas de ellas, es decir que la única manera de cambiarlas es modificando en el código fuente y recompilando el programa por completo. Aquí está la lista de estas variables y su

Constante	Valor	Comentario
Número de partículas	22	Para que se vea mejor el modelo, se elige un número par
Dimensiones caja	$0.75 \times 1.5 \times 1.5$	Se forma una caja de tapas cuadradas
Masa tela	10	La masa de cada partícula es entonces $\frac{10}{22^2}$
Radio pelota	0.25	
Posición pelota	(0, 1.5, 0)	Como el cuerpo neumático está centrado en el origen. La pelota está justo arriba de él en el centro
Δt	0.0025	Determinado experimentalmente

Cuadro 4.1: Valor de las constantes del experimento

significado.

Las variables que se decidieron dejar fijas, y que por ende se convierten en constantes en el experimento son las siguientes:

- El número de partículas por lado de la tela.
- Las dimensiones (alto, ancho y largo) de la caja.
- La masa total de la tela. Y por lo tanto, la masa de cada partícula es ésta cantidad dividida entre el número total de partículas.
- El radio y la posición de la pelota.
- El tamaño del paso en el tiempo Δt que se usa para los métodos numéricos de integración.

Los valores de estas constantes con las que se hizo este experimento están en el Cuadro 4.1.

4.1.1.2 Variables físicas del experimento

La variables físicas del experimento pueden ser modificadas en tiempo de ejecución por medio del menú que muestra en la Figura 4.2.

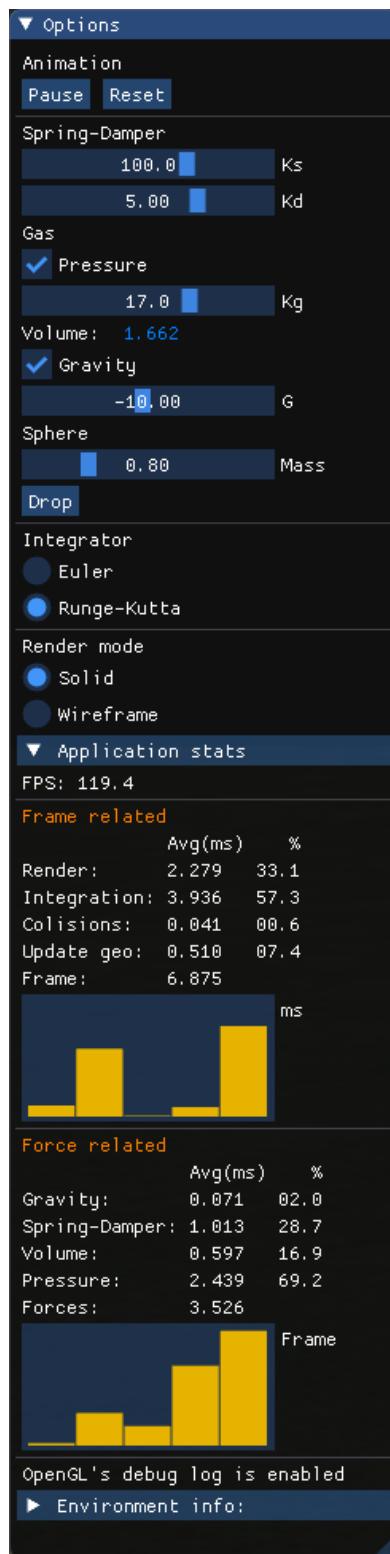


Figura 4.2: Menú de usuario.

Parámetro	Valor	Comentario
Masa	0.08	Masa del cuerpo rígido
Integrador	Runge-Kutta	Método Numérico con el que se integra.
Gas	Activado	El programa empieza con la fuerza del gas prendida
k_g	17	Constante de presión del gas
Gravedad	Activada	La gravedad está activada
g	-10.0	La gravedad es negativa (jala hacia abajo)
k_s	100	La fuerza de los resortes
k_d	5	El valor de damping

Cuadro 4.2: Valor inicial de los parámetros del experimento

Para la pelota o esfera (*Sphere*) que se refiere al cuerpo rígido. la única variable que se puede modificar es su masa.

Los parámetros del cuerpo flexible que es posible modificar se dividen en tres subcategorías una por cada fuerza que se acumula.

La categoría del *Gas*, al que se puede prender o apagar por medio del checkbox *Pressure*. Además, se puede modular su magnitud variando el valor de la constante k_g , por medio del control.

La de la *Gravity*, que al igual que la anterior se puede prender o apagar con el control y también se puede modular variando el valor de la constante g .

La última subcategoria es la debida al resorte amortiguador (*Spring-Damper*.) Esta fuerza no se puede apagar, pero se puede variar modificando dos parámetros k_s , que, como ya se dijo, controla la rigidez del resorte y k_d que controla el amortiguamiento o pérdida de energía debida al resorte.

Aunque no es propiamente una opción de la física. El menú también permite elegir que método numérico usar para integrar. Las opciones disponibles son: *Euler* y *Runge-Kutta*.

Los valores de default de estas variables (el valor que tiene al iniciar la ejecución) son los que muestra la Tabla 4.2.

4.1.1.3 Opciones de control y visualización

Hay otras opciones que se pueden modificar por medio del menú, que se refieren más a cómo controlamos la animación y a como se ve el modelo que a la física.

Dentro de las del flujo de la animación, sólo hay dos controles, el de *Pause/Play*, que puede detener (reanudar) la animación física (se sigue pudiendo mover la cámara). Y el botón de *Reset*, que devuelve a todos los objetos a su posición original.

El botón de *Drop*, hace que la esfera se deje caer.

Dentro de la categoría de *Render*, se encuentran las siguientes opciones:

La primera opción *solid*, usa el modelo de iluminación de Phong (los materiales se obtienen de texturas) para dibujar todos los objetos en la escena.

Y la segunda opción *wireframe*. Que solo cambia la manera de dibujar el cuerpo neumático. Dibuja con líneas azules las orillas de la caja y dibuja en una escala de color entre amarillo y rojo, los resortes (como líneas) y las partículas (como puntos) que forman el cuerpo flexible. El color asignado depende de la magnitud de la fuerza que actúa sobre ellos.

Finalmente, además del menú de usuario, hay opciones que solo se pueden modificar por medio del mouse y del teclado.

- Arrastrar el mouse con el botón izquierdo presionado, permite modificar el ángulo de la cámara. Se usa el modelo de *trackball camera*.
- La rueda del mouse, cambia el *zoom* de la cámara.
- Presionar la tecla “s” en el teclado toma un *screenshot*. Las imágenes son guardadas en el folder **Screenshoots** (Que debe ser creado por el usuario) en el mismo folder donde está el ejecutable.
- Presionar “Escape” en el teclado termina la ejecución del programa.
- Presionar la tecla “m” muestra/oculta el menú de usuario.
- Presionar “F11” cambia entre el modo de pantalla completa y de ventana.
- La tecla “p” es un atajo para el botón de *Play/Pause* de la animación.

- La tecla “r” es un atajo para el botón de *Reset* de la animación.

4.1.2 Características del entorno de pruebas

4.1.2.1 Software

El programa fue desarrollado en lenguaje C++ y los *shaders* que sirven para hacer el render fueron escritos en GLSL.

Para poder compilar el código fuente de este programa es necesario tener un entorno de programación que contemple lo siguiente:

Un compilador de C++ y las siguientes bibliotecas: OpenGL, glfw, Dear ImGui, GLM, GLEW, FreeImage y Assimp.

Todos éstos requerimientos son software libre (Más detalles de esto en el Apéndice). Aunque este trabajo fue desarrollado en su totalidad en un sistema operativo GNU/Linux, todos los requerimientos tienen la enorme ventaja de estar disponibles en cualquier plataforma. Si las bibliotecas están correctamente instaladas, el programa debe compilar y funcionar bajo cualquier sistema operativo.

4.1.2.2 Hardware

La mayoría de las pruebas fueron hechas en una laptop personal MSI GF65, con las siguientes características.

- Procesador: Intel Core i7-9750H CPU @ 2.60GHz × 12.
- Memoria: 32GB DDR2.
- Tarjeta de Video: Nvidia GeForce GTX 1660 Ti Mobile¹
- Sistema Operativo: Ubuntu 20.04 64 bits.

Esto no quiere decir que este sea el hardware mínimo, sólo que la *mayoría* de las pruebas se realizaron en éste hardware. Sin embargo, se ha ejecutado con éxito en

¹Aunque la laptop tiene la tarjeta de video descrita, el programa es tan simple que de hecho puede ejecutarse en la tarjeta de video integrada del procesador Intel sin problemas.

equipos bastante accesibles. Éste trabajo es de hecho una reedición, el trabajo original se desarrolló usando una laptop convencional en el año 2008. Actualmente, éste programa debe poder ejecutarse en cualquier computadora personal, de escritorio o equipo móvil sin problemas de capacidad del *hardware*. Hoy en día la única limitante (de existir) podría acaso ser el *software*.

4.2 Características físicas del modelo

Para poner en prueba la fidelidad del modelo, se hicieron una serie de pruebas. Estas pruebas tienen dos finalidades: primero probar la sensibilidad del programa ante la variación de sus parámetros físicos y segundo comprobar cualitativamente que el fenómeno físico funciona correctamente.

4.2.1 Probando la gravedad

La gravedad es la única fuerza que actúa tanto en el cuerpo flexible como en el cuerpo rígido. Para entender mejor cómo afecta se sugieren las siguientes pruebas.

Partiendo de los valores de *default*, se espera a que la tela se estabilice (Figura 4.3a). Ahora se *aumenta* la gravedad a su valor más *pequeño* (recordemos que hacer la gravedad más fuerte es hacerla más negativa), es decir, $g = -20$, se observa ahora cómo la gravedad es muy fuerte para que la presión del gas inflé la tela, por lo que queda colgando un poco. Las partículas que forman el cuerpo flexible son muy pesadas (son jaladas con más fuerza hacia abajo). Esta situación se muestra en la Figura 4.3b. Ahora se deja caer la pelota y se espera que se estabilice de nuevo el programa, (Figura 4.3c) la gravedad hace que la pelota y las partículas pesen más, sin embargo la fuerza del gas que no se ha tocado compensa de alguna manera y no deja que se hunda más la tela. Con la pelota estable sobre la tela, apagué la fuerza del gas. Al apagar la fuerza que equilibraba la gravedad todo se va hacia abajo, por la gravedad, pero como además esta fuerza es grande, la rigidez del resorte es poca para evitar un efecto de superelongación como el que se ve en la Figura 4.3d, donde la pelota es detenida por el piso.

Reinicie la animación y vuelva a los valores de *default*. De nuevo espere a que se estabilice la tela ahora disminuya la gravedad a su máximo valor posible, es decir una gravedad

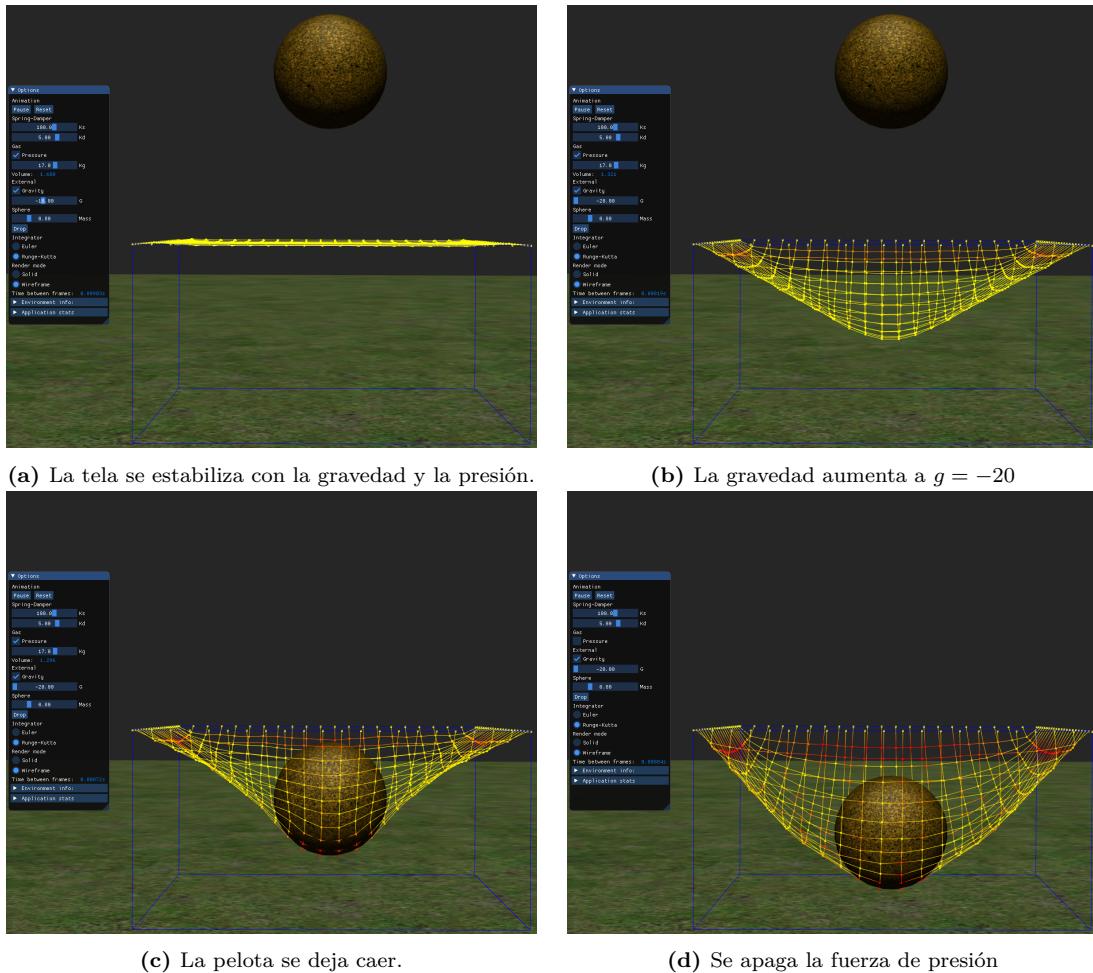
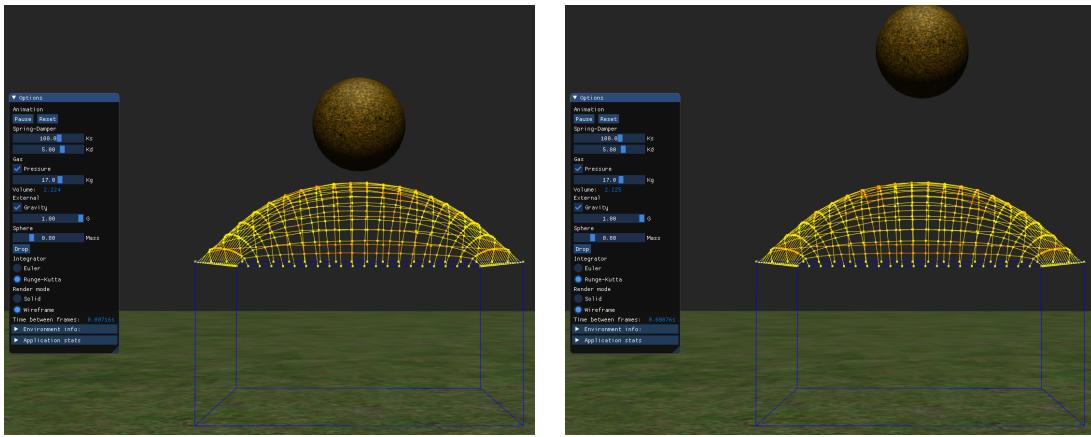


Figura 4.3: Probando la fuerza de gravedad



(a) La presión y la gravedad jalan la tela hacia arriba. (b) La pelota también es jalada hacia arriba

Figura 4.4: La fuerza de gravedad cambia de signo

positiva: $g = 1$. Ahora verá que la tela se va más hacia arriba: se infla más el cuerpo flexible. Esto se debe a que ahora la gravedad no se opone a la presión del gas, sino más bien le favorece, por lo que el cuerpo flexible, es jalado hacia arriba aún más, como se ve en la Figura 4.4a. Ahora deje caer la pelota, como la gravedad es positiva y pequeña (su valor absoluto en una décima parte del valor normal) la pelota *sube lentamente*, y se aleja del cuerpo flexible (Figura 4.4b). La constante de gravedad influye en la velocidad de caída de los cuerpos.

Por último vamos a reiniciar la animación y partiendo de los valores de *default* de los parámetros, se apaga la gravedad. Este comportamiento equivale a hacer $g = 0$ con la diferencia de que se hacen menos cálculos, ahora vemos cómo de nuevo la tela se mueve hacia arriba, cómo la gravedad se oponía a la fuerza del gas y ya no está, el gas empuja la tela aún más hacia arriba. Si en esta situación se deja caer la pelota no pasará nada, debido a que la caída de la pelota *depende de la gravedad*, y al no haber, simplemente no hay caída.

Si primero se deja caer la pelota sobre la tela, y una vez que ésta se estabiliza se apaga la gravedad, la pelota es empujada fuera del cuerpo flexible por un impulso. Ver la Figura 4.5.

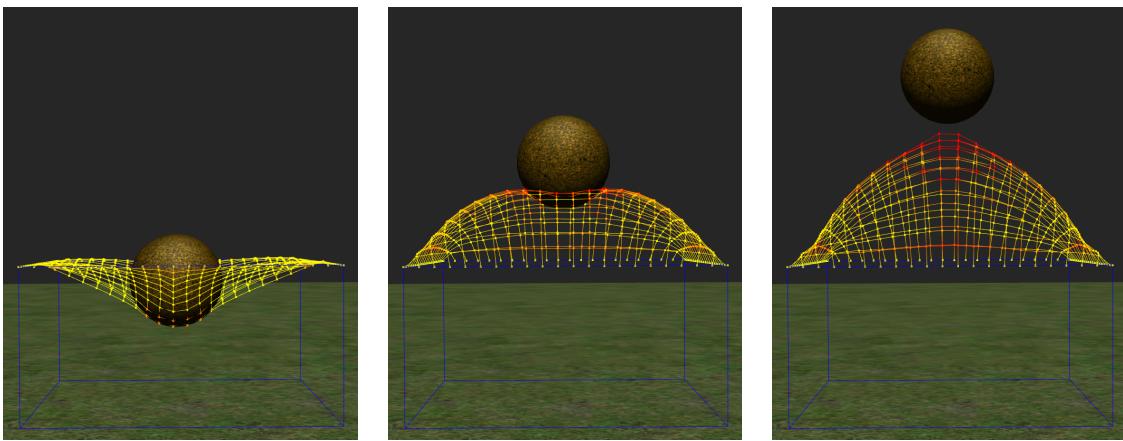


Figura 4.5: La fuerza de gravedad es apagada después de que la pelota descansa sobre la tela.

4.2.2 Probando la fuerza del gas

Ahora se harán pruebas sobre la fuerza del gas. La fuerza del gas, por el diseño de nuestro experimento, se opone a la fuerza de gravedad, y actúa sólo sobre el cuerpo flexible. Sin embargo, tiene cierto efecto sobre la velocidad de las partículas del cuerpo flexible, que a su vez tienen cierto efecto sobre el cuerpo *rígido* cuando hay una colisión.

Inicie el programa con los valores de *default*, ahora apague la fuerza del gas y espere a que se estabilice el modelo, como se ve en la Figura 4.6a. Como no hay fuerza del gas, la tela o cuerpo flexible cuelga agarrada de las orillas de la caja. Ahora deje caer la pelota sobre el cuerpo flexible y espere a que se estabilice la animación, justo como se ve en la Figura 4.6b. Prenda la fuerza del gas y observe cómo la pelota es lanzada hacia arriba súbitamente como se ve en las Figuras 4.6c y 4.6d.

Otra prueba es ver los efectos de la variación de la constante k_g . Inicie la animación con los valores de *default* y haga la constante k_g pequeña, por ejemplo $k_g = 11$; verá cómo el gas no es lo suficientemente fuerte para inflar el cuerpo flexible. Ahora aumente poco a poco la constante k_g y observe cómo se infla cada vez más el cuerpo flexible. En la Figura 4.7 se ilustran estas situaciones para diferentes valores en aumento de la constante k_g .

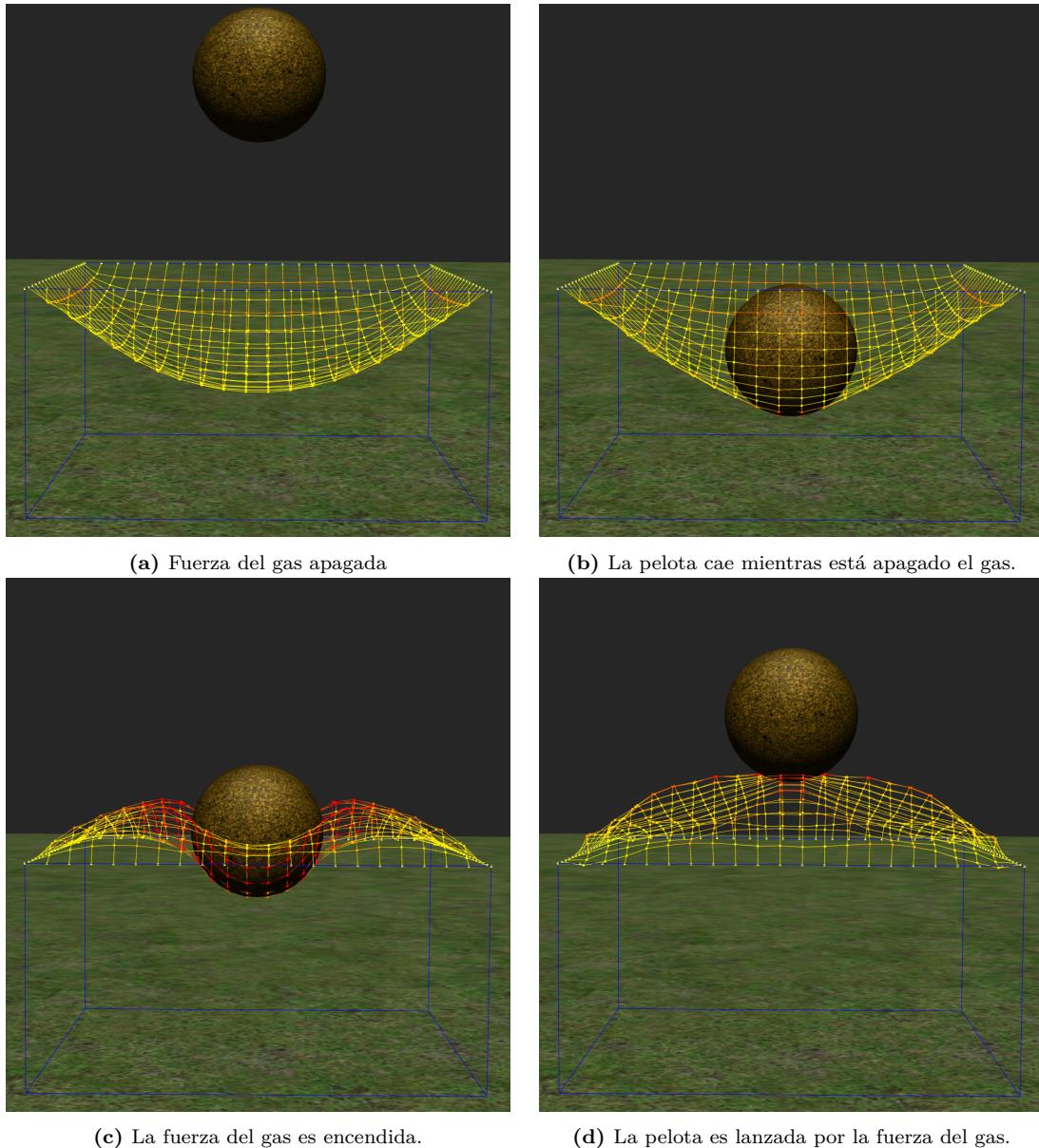
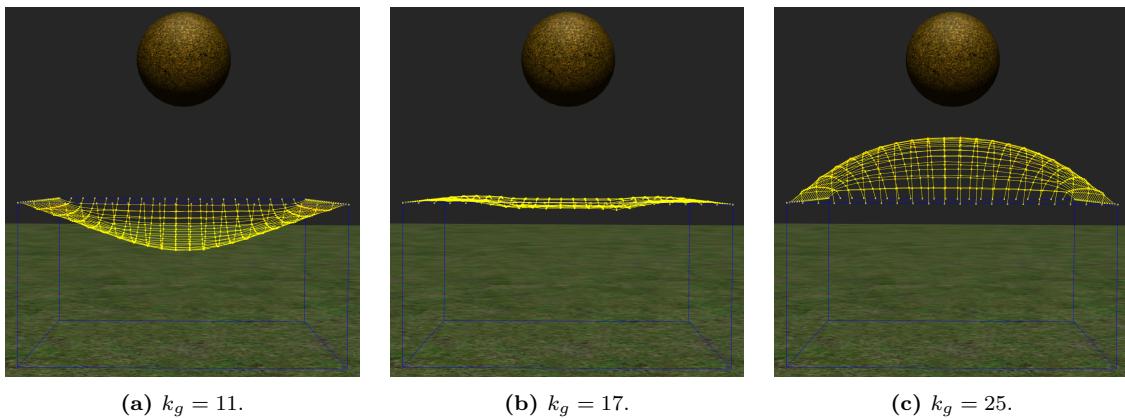


Figura 4.6: La fuerza del gas interactúa con el cuerpo rígido.

(a) $k_g = 11$.(b) $k_g = 17$.(c) $k_g = 25$.**Figura 4.7:** El parámetro k_g cambia la fuerza del gas.

4.2.3 Probando los resortes amortiguadores

Las siguientes pruebas sirven para mostrar cómo funcionan los resortes amortiguadores. Primero vamos a hacer una prueba con la constante de *damping* k_d . Como ya se dijo, el damping es una forma de perder energía del sistema y, por lo tanto, de eventualmente estabilizarse. Iniciemos la animación con los valores de *default* y hagamos la constante $k_d = 0$, es decir quitemos todo el *damping*. Vemos que la tela empieza a oscilar rápidamente como consecuencia del gas. Ahora quitemos también el gas y esperemos un momento; veremos como la tela oscila sin detenerse ni estabilizarse en ningún momento, es cierto que cada vez oscila menos, pero ciertamente tardará mucho en detenerse (o nunca se detendrá).

En la Figura 4.8, podemos ver diferentes oscilaciones sin control por falta de un amortiguador. Como ya se había dicho, el amortiguador agrega realismo (en la total ausencia de *damping*, el cuerpo flexible se ve poco real). La contra parte es que a mayor *damping*, también hay menos estabilidad numérica, por lo que el modelo es sumamente sensible al aumento en este parámetro.

Podemos ir aumentando el valor de k_d poco a poco y ver cómo el modelo se vuelve inestable. Si se pone el máximo posible $k_d = 7$ y se reinicia la animación, el modelo explota (ver Figura 4.9).

Ahora analicemos la constante de rigidez k_s . Esta constante hace a los resortes más poderosos, por lo que en general hace que las partículas que están unidas por ellos, se separen menos, es decir hace más estable el cuerpo flexible en general, además de

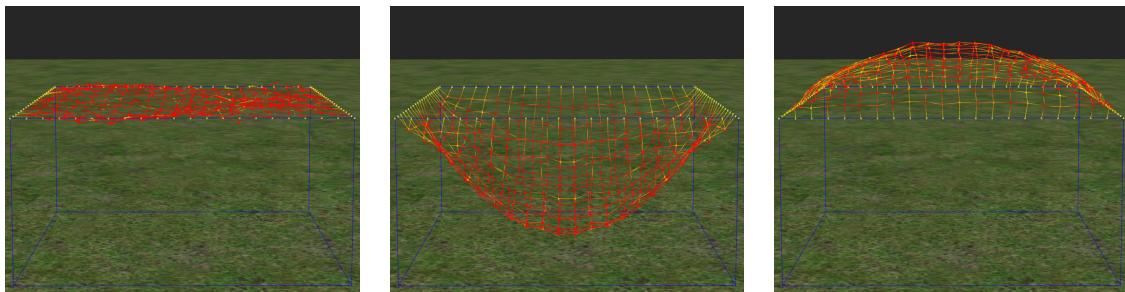


Figura 4.8: Oscilación sin control: $k_d = 0.0$

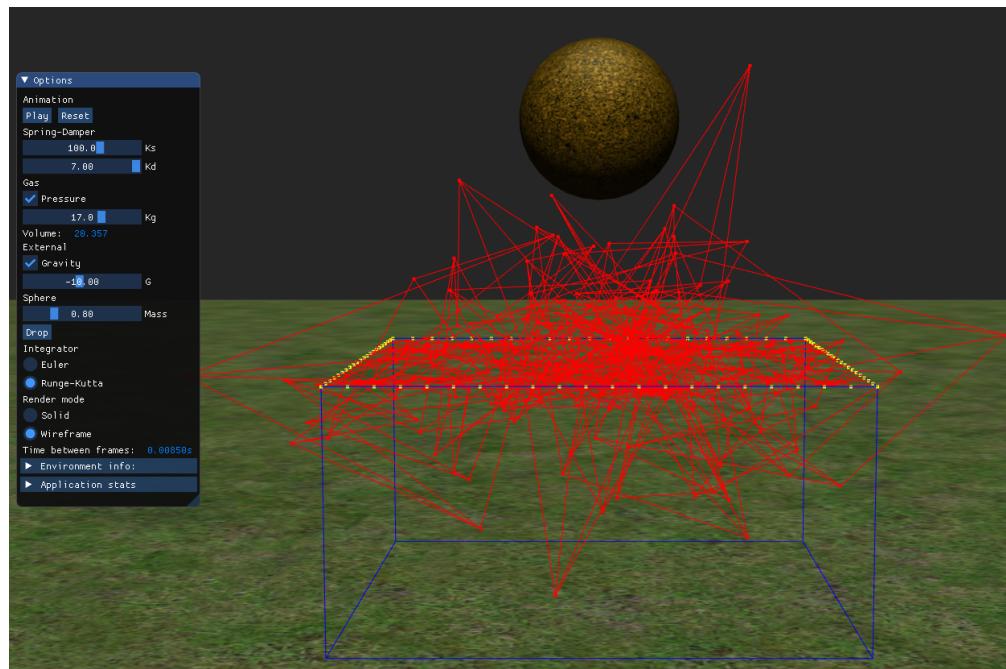


Figura 4.9: La animación explota por inestabilidad numérica

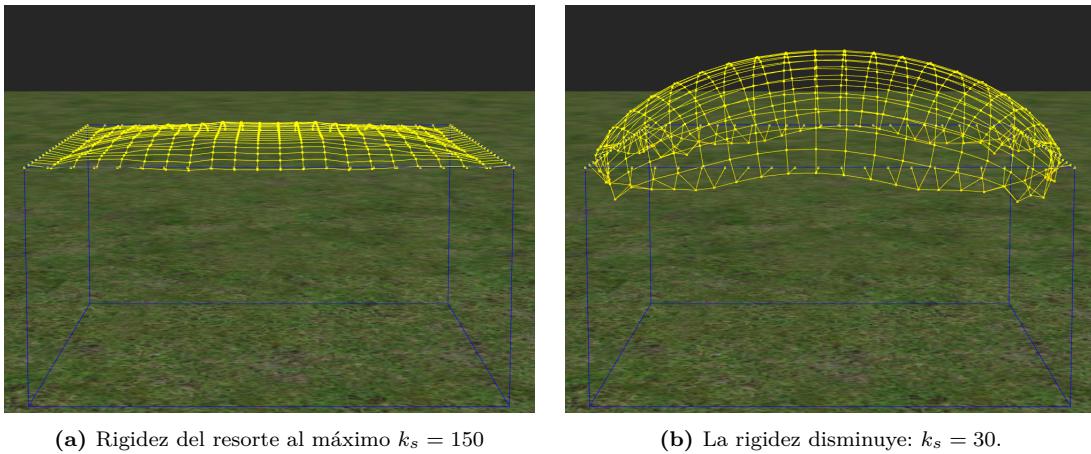


Figura 4.10: La fuerza del gas es constante a $k_g = 17$ y variamos k_s

prevenir el efecto de super elongación.

Iniciamos la animación con los valores de default y pongamos la constante del resorte $k_s = 200$ es decir a su máximo valor. Ahora apaguemos el gas y vemos como la caída de la tela es menor, sin embargo al apagar y prender varias veces el gas también nos damos cuenta de que el cuerpo flexible parece oscilar más, como consecuencia de que los resortes en general jalan más fuerte, tanto hacia arriba como hacia abajo.

Ahora con la constante $k_g = 17$, vayamos poco a poco subiendo la presión del gas. Podemos ver cómo aun con $k_g = 17$, si $k_s = 150$ el cuerpo flexible se expande poco; esperemos a que se estabilice, como se ve en la Figura 4.10a. Ahora que tenemos el modelo estable y con k_s máximo, poco a poco vayamos haciendo k_s más pequeña y podremos apreciar cómo el cuerpo parece inflarse más (la resistencia de la membrana que lo mantiene unido es menor), como se ve en la Figura 4.10b, hasta llegar al momento donde explota ($k_s = 0.0$).

4.3 Análisis del desempeño

Una de las métricas más usadas para medir el desempeño de un programa gráfico es el número de veces que se actualiza el *frame buffer* en un segundo. Esta medida es conocida como *fps* que es la abreviatura en inglés de *frames per second*. En general, es aceptado que para que una aplicación gráfica se considere en tiempo real (como debe ser un videojuego) se debe de alcanzar al menos 60 fps.

El programa es tan simple que aun en una tarjeta de video integrada en el CPU alcanza alrededor de 120 fps². Por lo tanto, aunque esta medida es calculada (Figura 4.2) en realidad nos dice muy poco del desempeño del programa.

Por esta razón, decidí utilizar el tiempo del proceso del CPU (medido en milisegundos) para hacer el análisis siguiente.

4.3.1 Desempeño del programa

Vamos a medir el tiempo que el CPU tarda en hacer todos los cálculos para producir un *frame*. Vamos también a cronometrar el tiempo que toma procesar varias etapas del programa. Tomando la primera medida como base, vamos a calcular la proporción del tiempo que es usado en cada etapa.

Decidí cronometrar las siguientes etapas:

Tiempo entre frames: La medida base. Este es el intervalo de tiempo que toma hacer una iteración del ciclo principal del programa. Es decir el ciclo mostrado en la Figura 2.2).

Tempor de render: El tiempo que toma presentar al *frame buffer* una vez que se tienen los datos actualizados de todas las partículas y demás objetos de la escena al GPU. Es decir, el tiempo que toma la ejecución de todos los *pipelines* gráficos.

Tiempo de interacción: Es el tiempo que toma calcular la posición y velocidad de cada partícula. En otras palabras el tiempo que toma ejecutar RK4.

Tiempo de manejo de colisiones: El tiempo que toma ejecutar la detección y respuesta a las colisiones.

Tiempo de actualización de la malla: El tiempo que toma primero recalcular las normales y después transmitir los datos completos (posiciones y normales de cada partícula) al GPU.

²De hecho, cuando se ejecuta el programa usando la tarjeta de video Nvidia, el driver gráfico *alenta* el programa para sincronizarse con la tasa de refresco del monitor. Es decir, al darse cuenta de que el programa produce frames más rápido de lo que los puede mostrar el hardware; hace el programa más lento para no desperdiciar recursos. Una tecnología conocida como *vsync*

Adicionalmente, una parte del el *tiempo de integración* es utilizado en evaluar la función fuerza. Una evaluación si se integra con Euler, o cuatro evaluaciones si se integra con RK4. Para ser más específicos y hacer un análisis parecido al anterior, se tomaron cuatro medidas además de la medida base. Para la siguientes métricas se calcularon los tiempos totales que se pasa haciendo evaluaciones de cada parte de la función fuerza. Es decir, que en el caso de que se integre con Euler es una medida, en el caso de que se integre con RK4 la suma de cuatro mediciones.

Tiempo de evaluación de \mathbf{F} : La medida base. Este es el tiempo total que se pasó evaluando la función \mathbf{F} en cada frame.

Gravedad: El tiempo total que tomó acumular la \mathbf{F}_g a todas las partículas.

Resortes: El tiempo total que tomó acumular \mathbf{F}_s Y \mathbf{F}_d usando todos los resortes amortiguadores.

Volumen: El tiempo total que se pasó calculando V , el volumen del cuerpo neumático. Este tiempo está contenido en el tiempo de presión. Ver de nuevo el Listado 3.7.

Presión: El tiempo total que se pasó acumulando \mathbf{F}_p recorriendo todas las caras.

Para hacer las mediciones se fijaron las siguientes condiciones: Hacer render con caras sólidas y texturas. Utilizar el método de RK4 para integrar. Dejar caer la pelota sobre la tela para forzar que haya colisiones en cada frame. Desde luego, que los tiempos anteriores son diferentes para cada frame. Por esta razón se decidió ejecutar el programa y registrar los tiempos de cada frame de la ejecución.

Se registraron alrededor de 3200 frames. Como primer método de exploración de los datos se presenta el *resumen de 5 números* de cada una de las categorías. El resumen de cinco números nos ayuda a ver que tanta dispersión hay entre los datos y que forma puede tener la distribución. Está formado por cinco estadísticos: El mínimo de los datos (min), el primer cuartil (Q_1), la mediana (\tilde{x}), el tercer cuartil (Q_3) y el máximo de los datos (max).

Como puede verse en el Cuadro 4.3 y en el Cuadro 4.4 hay algunos *outliers* en la parte alta de los datos. Ambas distribuciones están sesgadas a la derecha. Esto quiere decir

	Render	Integración	Colisión	Malla	Frame
min	0.2533	2.2235	0.0257	0.3272	3.1157
Q_1	0.9312	2.6941	0.0325	0.3957	6.3076
\tilde{x}	2.9606	3.7973	0.0442	0.5198	7.5509
Q_3	5.1519	5.1556	0.0572	0.6769	8.3865
max	26.8164	12.3506	0.1548	1.7716	30.3281

Cuadro 4.3: Resumen de cinco números de las medidas para calcular un frame.

	Gravedad	Resortes	Volumen	Presión	F
min	0.0383	0.5527	0.3331	1.3756	1.9747
Q_1	0.0475	0.6788	0.4072	1.6615	2.3912
\tilde{x}	0.0677	0.9531	0.5679	2.3528	3.3860
Q_3	0.0950	1.3141	0.7834	3.2024	4.6050
max	0.2386	3.1087	1.9180	7.7122	11.0151

Cuadro 4.4: Resumen de cinco números de las medidas relativas a la evaluación de **F**.

	Render	Integración	Colisión	Malla	Frame
\bar{x}	3.0502	4.4734	0.0498	0.5878	8.2945
%	36.77	53.93	0.60	7.09	

Cuadro 4.5: Proporción de tiempo de los cálculos con respecto a un frame.

	Gravedad	Resortes	Volumen	Presión	F
\bar{x}	0.0803	1.1376	0.6796	2.7712	3.9920
%	2.01	28.50	17.02	69.42	

Cuadro 4.6: Proporción de tiempo de los cálculos con respecto a la evaluación de F.

que hay algunos frames particularmente lentos ³.

Sin embargo, en realidad los datos no están tan dispersos pues las distancias entre los Q_1 y Q_3 es proporcionalmente pequeña en todos los casos. Con esto en mente, decidí tomar la media aritmética (\bar{x}); mejor conocida como el *promedio*, de cada medida para poder hacer el cálculo que nos interesa.

Para calcular la proporción que cada medida representa con respecto de la medida base, simplemente dividió el promedio de la medida entre el promedio de la medida base y lo multiplique por 100 (Para expresarlo en forma de porcentaje)

Hay algunos datos interesantes en el Cuadro 4.5. Como se esperaba, la rutina computacionalmente más demandante es el método numérico, con cerca del 54% del tiempo. Resalta también que el manejo de las colisiones es casi insignificante pues no alcanza ni siquiera un 1%. Esto puede deberse a que en realidad el número de partículas es pequeño y la detección entre punto y esfera es muy sencilla.

Finalmente, quiero señalar que dado que hay más operaciones que no se midieron. El manejo de la interacción con el usuario es uno, por dar un ejemplo. Por esa razón, la suma de los componentes no da el 100% del tiempo para producir un frame.

³Esto es común en los programas gráficos, un frame puede ser inusualmente lento por varias razones. Por ejemplo: si la ventana cambio de tamaño (y hubo que recrear el *frambuffer*), si hubo un cambio en el modo de *render* (y hubo que recrear el *pipeline* gráfico) o si hubo que cambiar *assets* (y hay que actualizar la memoria de video)

Con respecto al Cuadro 4.6 vemos que el cálculo de la \mathbf{F}_p es el más costoso con el 70% del tiempo. Seguido de los cálculos de \mathbf{F}_s y \mathbf{F}_d con el 28.5%. La acumulación de \mathbf{F}_g es casi despreciable. Esto quiere decir, que el tiempo de cálculo es directamente proporcional al número de partículas involucradas. Hay *una* partícula involucrada para acumular \mathbf{F}_g , *dos* para acumular \mathbf{F}_s y \mathbf{F}_d y *cuatro* para acumular \mathbf{F}_p . Sin mencionar, que para calcular \mathbf{F}_p , primero debemos calcular V .

El tiempo de cálculo de V forma parte del tiempo de cálculo de \mathbf{F}_p . De hecho, del 69% de tiempo que ocupamos en calcular \mathbf{F}_p , el cálculo de V aporta un 17% (casi una cuarta parte). Es por esta razón que si se suman los porcentajes de gravedad, resortes y presión casi se da el total para producir \mathbf{F} .

Por último hago énfasis en que aún cuando cada experimento con del programa arrojará datos diferentes, las proporciones reportadas en el Cuadro 4.5 y el Cuadro 4.6 tienden a ser *casi* las mismas. Puede volverse a mirar la Figura 4.2 que se capturó con otra ejecución del programa y también reporta proporciones parecidas. Mas aún, cualquier implementación que siga los algoritmos descritos en éste trabajo *deberá* presentar proporciones muy similares en los cálculos. Por lo tanto, los datos en los Cuadros 4.5 y 4.6 son una aportación relevante de ésta tesis.

Conclusiones

Después de haber hecho pruebas y evaluado los resultados obtenidos presentó las siguientes conclusiones.

La más importante: las ecuaciones diferenciales, permiten modelar gráficamente el comportamiento físico de un cuerpo neumático que interactúa con un cuerpo rígido en tiempo real. También hay que recalcar que este modelo tiene sustento en la física. Es decir es un modelo basado en física y no en geometría.

Creo también que la técnica aquí usada; la de utilizar un gas ideal dentro del cuerpo flexible, es muy buena para enfrentar este tipo de problemas. La técnica nos proporciona todo lo que desearíamos para hacer una animación: es computacionalmente barata; al menos lo suficiente para alcanzar tiempo real, y es físicamente adecuada para modelar el gas.

La implementación como se describe, depende poco del poder gráfico. El render sigue siendo una operación relativamente barata *en comparación* con los cálculos numéricos. Dentro de los cálculos numéricos, lo que más contribuye es el método numérico de Runge Kutta seguido de la acumulación de la fuerza del gas.

Si bien el utilizar el método de Euler hace considerablemente más rápido de ejecutar el programa, recomiendo usar el método de Runge Kutta. Esto ofrece una mayor estabilidad ante la variación de las constantes, lo que se traduce en la posibilidad de una mejor interacción. También creo que este método no es tan complejo de implementar.

El *damping* es un parámetro que debe de estar presente. Sin embargo, requiere tener mucho cuidado al determinar un valor adecuado. Considero que el valor más adecuado es el más grande posible sin que la animación explote. Este modelo es muy sensible a las variaciones (incluso *pequeñas*) de éste parámetro.

Se recomienda ampliamente que al momento de programar se tome tiempo para hacer una adecuada elección de las estructuras de datos. Guardar todo en una estructura de datos lineal simplifica bastante la programación. De igual manera, recomiendo tener diferentes maneras de acceder a las propiedades de las partículas, ya sea por los resortes o por medio de las caras.

El modelo del gas ideal es un modelo muy recomendado para simular cuerpos flexibles. Con una adecuada elección de los parámetros se puede tener un comportamiento bastante realista. Por lo que considero que el objetivo fundamental de este trabajo se cumple.

Hay muchas mejoras posibles para esta implementación, pienso que hay campo para futuras investigaciones en los siguientes detalles:

- Investigar sobre un método numérico más eficiente, aquel que dé más rapidez sin perder estabilidad.
- El manejo de colisiones más eficiente.

Para el método numérico se hicieron pruebas con el integrador de Verlet. Sin embargo, abandoné ese camino porque en un esquema como ése no se toma en cuenta la velocidad de la partícula. Por lo que no encontré manera de responder a las colisiones. Creo que investigar la manera de incluir una fuerza de impulso como respuesta a la colisión y utilizar el integrador de Verlet daría resultado.

Hay también que considerar una respuesta a las colisiones que conlleven una pérdida de energía al momento de la colisión. Es decir, eliminar el supuesto de colisiones perfectamente elásticas. Aunque escribí en el segundo capítulo sobre ellas, no fueron implementadas en el programa.

Mi forma de manejar tanto la detección como la respuesta de las colisiones no es la más eficiente, pues pruebo una a una las partículas del cuerpo flexible. Alguna estructura de datos geométrica (como un *octree* o un *range tree*) que probara sólo las partículas cercanas haría mejor la detección.

También hay que considerar que el cuerpo flexible puede colisionar consigo mismo. Es decir, hace falta probar colisiones de las caras del cuerpo con cada una de las partículas y luego responderlas. Considero que en cualquier configuración del cuerpo flexible que no sea un volumen convexo éste problema estará presente.

Apéndice: Sobre el software libre

Un objetivo secundario cuando empecé a hacer esta tesis, fue que toda ella fuera hecha con *software libre*, o al menos *open source*. El software libre es aquel que cumple con las cuatro libertades definidas por la Free Software Foundation (www.fsf.org). La libertad de usarlo para cualquier propósito, la libertad de estudiarlo y adaptarlo, la libertad de distribuir copias y la –polémica– libertad de distribuir las mejoras. Si el software en cuestión sigue las tres primeras libertades, pero no la última es usualmente considerado *open source*. En este sentido, todo software libre es *open source*, pero el *open source* no es necesariamente software libre ⁴.

Para ésta segunda edición, éste objetivo finalmente se cumplió. Toda la tesis y el programa fueron desarrollados con software libre u *open source*.

Para empezar se utilizó gcc (gcc.gnu.org) como compilador de C++. Para acceder a OpenGL (www.opengl.org) se usó el driver libre de Intel (01.org/linuxgraphics). Aunque también hice pruebas usando el driver propietario de Nvidia (www.nvidia.com), éste uso fue más una curiosidad que una necesidad y bien pudo haberse omitido.

Para resolver las funciones de OpenGL (es decir como *extension loader*), se uso la biblioteca GLEW (glew.sourceforge.net). Como biblioteca para la creación de ventanas y de interacción con el usuario utilicé GLFW (www.glfw.org). Para cargar y escribir imágenes usé FreeImage (freeimage.sourceforge.io). Para el menú de usuario se usó la biblioteca Dear ImGui (github.com/ocornut/imgui). Para hacer mi vida mucho más sencilla, la biblioteca GLM (github.com/g-truc/glm) cubrió mis necesidades de vectores, matrices y álgebra lineal en general.

El texto de la tesis se escribió en L^AT_EX, con la implementación de TeX Live (tug.org/texlive).

⁴Admito que esta afirmación no es del todo correcta. Pero pragmáticamente puede pensarse así y no deseo entrar en debates innecesarios que tanto han dañado a ambas comunidades.

La gráfica de la Figura 1.5 fue hecha en el lenguaje de programación R (<https://www.r-project.org/>). Los diagramas de la Figura 2.2 y la Figura 3.1 fueron hechos en Dia (<http://dia-installer.de>). El resto de las figuras (que no son *screenshots* de mi programa) fueron hechas usando el editor de gráficos vectoriales Ipe (<ipe.otfried.org>). Cuando tuve la necesidad de hacer edición de imágenes tipo *raster* (es decir, con pixeles) use a veces Gimp (<www.gimp.org>) y a veces ImageMagick (<imagemagick.org>), dependiendo de que acomodara mejor la situación.

Todo el desarrollo se hizo en un sistema operativo GNU/Linux, en particular con la distribución Ubuntu (<ubuntu.com>). Para editar el texto usé Kile (<kile.sourceforge.io>) y lo recomiendo ampliamente. Como IDE para desarrollar y depurar el programa utilicé Eclipse CDT (<www.eclipse.org/cdt>) y cubrió todas mis necesidades. El control de versiones fue hecho en git (<git-scm.com>) y como servicio de repositorios use GitHub (<github.com>). Esto último me permitió trabajar en más de una computadora cuando lo requerí y me dio la tranquilidad de tener siempre respaldos en la nube.

Por último, hago disponible también a través de GitHub (<nemediano/TesisLicenciatura>) tanto el texto total en L^AT_EX de éste trabajo como el código fuente del programa. En repositorios separados, también he creado una plantilla para hacer tesis de la UNAM y una plantilla para hacer aplicaciones gráficas con OpenGL. Este trabajo es un ejemplo del uso de ambas.

Modelado Gráfico de un Cuerpo Neumático con OpenGL a Base de Ecuaciones Diferenciales © 2008 por Jorge Antonio García Galicia está licenciado bajo Attribution 4.0 International.

Para ver una copia de esta licencia, visite <creativecommons.org/licenses/by/4.0/>

Bibliografía

- [1] David Baraf y Andrew Witkin. «Phisically Based Modeling: Principles and Practice». En: *Sigraph* (1997). Las notas de un curso que se llevó a cabo en Siggrad, en 1997, bastante completas aunque requieren un buen nivel de programación en C para poder entenderlas. Tiene un error en la fórmula para separar velocidades en componentes ortogonales.
- [2] Paul Blanchard, Robert L. Devaney y Glen R. Hall. *Ecuaciones Diferenciales*. 1^a ed. International Thompson Editores, 1999.
- [3] David M. Bourg. *Physics for Game Developers*. Inglés. 1^a ed. Un libro muy curioso que nos ayuda a entender la física y pasarlala a código de una manera muy elegante, hay muchísimo material en este libro y para este trabajo sólo se utilizó un par de capítulos. California, EU: O'Reilly, 2002.
- [4] William E. Boyce y Richard C. DiPrima. *Ecuaciones Diferenciales y Problemas con Valores a la Frontera*. 4^a ed. Un clásico de Ecuaciones diferenciales y cubre a detalle muchísimos temas, yo lo utilicé primordialmente para el análisis del error de los métodos numéricos. Limusa Willey, 2002.
- [5] María del Carmen Villar Patiño. «Comparación de Implementaciones en HW y SW de Métodos de Integración Numérica para Simulación de Tela». La tesis de Maestría de mi asesora de tesis, muy útil en la parte de las colisiones y de la integración. Tesis de mtría. Instituto Tecnológico y de Estudios Superiores de Monterrey, 2003.
- [6] Gabriel Valiente Ferlugo. *Composición de Textos Científicos Con LATEX*. 1^a ed. Un libro sobre la edición de textos científicos que fue utilizado para escribir este trabajo. México DF, México: Alfaomega, 2001.

- [7] Glenn Fielder a.k.a Gaffer. *Game Physics Articles*. Un artículo con tips sobre física para videojuegos, muy relajado y muy útil para empezar a adentrarse en estos temas, me ayudó en la implementación de los métodos de integración. 2005.
- [8] Tomas Jakobsen. «Advanced Character Physics». En: *IO Interactive* (2001). Un artículo muy útil donde explican cómo modelar por medio de resortes y se usa el integrador de Verlet. Además lo escribió uno de los creadores del juego de Hitman, a partir de las técnicas con las que se desarrollo el juego.
- [9] Danny Kodicek. *Mathematics and Physics for Programmers*. Inglés. 1^a ed. Un libro muy bueno para entender la física necesaria para hacer videojuegos, aunque empieza desde aspectos muy básicos, como álgebra y cálculo, toca muy bien el tema de las colisiones y fue una de las fuentes más importantes al escribir esta tesis. Massachusetts, EU: Charles River Media, 2005.
- [10] Maciej Matyka. *How to Implement a Pressure Soft Body Model*. Un articulo muy bien explicado sobre como implementar un modelo de presión en un cuerpo flexible, ideal para comenzar a leer sobre el tema y fue punto clave en esta tesis. 2004.
- [11] Maciej Matyka y Mark Ollila. «Pressure Model of Soft Body Simulation». En: *Proc. of Siggrad* (2003). El primer artículo que leí sobre el tema y que me motivó a hacer esta tesis, muy recomendable junto con el segundo articulo.
- [12] Joan Trias Pairo. *Geometría para la informática Gráfica y el CAD*. 1^a ed. Libro de geometría, muy útil como referencia trata el tema con mucha seriedad, por lo que incluye numerosas demostraciones, también asume que el lector es proficiente en Álgebra Lineal. Las fórmulas del cálculo de volumen están muy basadas en este texto. Madrid, Espana: AlfaOmega, 2005.
- [13] Xavier Provot. «Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth Behavior». En: *INRIA* (1995). Otro articulo de modelado de ropa, que propone una técnica interesante para prevenir el efecto de súper elongación.
- [14] Robert Resnick, David Halliday y Kenneth S. Krane. *Fisica*. 4^a ed. Un clásico de física, nunca puede faltar y fue consultado para entender un poco mas de los fenómenos analizados. México DF, México: CECSA, 1999.

- [15] Shepley L. Ross. *Introducción a las Ecuaciones Diferenciales*. 3^a ed. Un libro muy sencillo de Ecuaciones Diferenciales, yo lo utilice en mi primer curso para está materia y aun lo consulto muy a menudo, es sencillo y es excelente para aprender. Interamericana, 1985.
- [16] Dave Shreiner, Mason Woo y Tom Davis. *OpenGL Programming Guide*. Inglés. 4^a ed. El libro clásico de OpenGL, es escrito por los mismos desarrolladores de la API, y nunca está de mas. Boston, EU: Addison Wesley, 2004.
- [17] B. Spanlang T. Vassiliev e Y. Chrysanthou. «Fast Cloth Animation on Walking Avatars». En: *EuroGraphics 2001* 20.3 (2001). Un artículo de implementación de modelos de ropa, lo leí por una técnica que usan para prevenir el efecto de súper elongación, y para entender por que a mi no me afecta dicho fenómeno., pág. 8.
- [18] Richard S. Wright y Benjamin Lipchak. *Programación OpenGL*. 4^a ed. El primer libro de OpenGL, que existe en nuestro idioma en el país, es muy extenso y la única queja es que la traducción deja un poco que desear. Madrid, España: Anaya Multimedia, 1999.
- [19] Cha Zhang y Tsuhan Chen. «Efficient feature extraction for 2D/3D objects in mesh representation». En: *Proceedings 2001 International Conference on Image Processing (Cat. No.01CH37205)*. Vol. 3. Un artículo muy útil donde explican cómo hacer cálculos geométricos de mallas en 2 y 3 dimensiones. 2001, 935-938 vol.3. DOI: [10.1109/ICIP.2001.958278](https://doi.org/10.1109/ICIP.2001.958278).