UNIVERSIDAD DE MURCIA, FACULTAD DE MATEMÁTICAS

DEPARTAMENTO DE MATEMÁTICAS

Trabajo Fin de Grado

Matemáticas Aplicadas a la Realidad Virtual

Autor:
José Andrés Muñoz

Supervisado por:
Eliseo Chacón Vera

5 de septiembre de 2016

Declaración de Originalidad

JOSÉ ANDRÉS MUÑOZ, autor del TFG

"Matemáticas Aplicadas a la Realidad Virtual"

bajo la tutela del profesor ELISEO CHACÓN VERA, declara que el trabajo que presenta es original, en el sentido de que ha puesto el mayor empeño en citar debidamente todas las fuentes utilizadas.

En Murcia, a 5 de septiembre de 2016.

(Nota: En la Secretaría de la Facultad de Matemáticas se ha presentado una copia firmada de esta declaración).

Agradecimientos

Agradecer en primer lugar a mi familia pues, si hoy estoy donde estoy, no es gracias sino a ellos que por su gran esfuerzo, en todos los sentidos, me han apoyado y animado a seguir cada día, motivándome para poder conseguir mi sueño, que hoy estoy cumpliendo. Y junto a ellos, todas las personas que, estando a mi lado, saben el sudor que ésto me ha conllevado.

Agradecer, por supuesto, a mi tutor Eliseo, por haberme dado la oportunidad de haber podido realizar este trabajo y atenderme siempre con la paciencia que lo ha hecho. Dar también, cómo no, mi más sincero agradecimiento a mis dos profesores ya que, sin tener por qué, ahí estuvieron siempre con su mano tendida para este trabajo: Gregorio, por la paciencia y la gran ayuda desinteresada ofrecida en los incontables y extensísimos correos que intercambiábamos sobre la programación; y Salvador, por tener siempre la puerta abierta a cualquier ayuda que pudiera dar, tanto académica para la propia redacción de este trabajo, como personal. Y, finalmente, no quisiera terminar sin dar una mención especial de agradecimiento a mi gran estimado Alberto del Valle, por su gran dedicación tanto a la Facultad como a sus alumnos, y quien siempre ha tenido abierta la puerta de su despacho, en particular, con un cariño especial para algunos de nosotros. Y hacer, de igual modo, una mención especial, en este trabajo, a mi antigua profesora, y amiga, Virtudes, por su gran granito de arena y su siempre inestimable ayuda a pesar de los años que van pasando.

Gracias a todos.

Índice general

Introducción

"Las matemáticas son el alfabeto
con el cual Dios ha escrito el universo."
— Galileo Galilei.

Las matemáticas dominan la vida. Tras cuatro años de carrera, cualquier estudiante de matemáticas conoce la trascendencia que éstas presentan en el mundo. El universo, y todo lo que nos rodea, está regido por leyes físicas las cuales están dominadas por las primeras, quienes sustentan su base. Todo proceso físico del cual podamos ser conscientes cada día tiene un principio matemático detrás. Si esto es así, podemos llegar a preguntarnos hasta qué punto somos capaces de conocer dichos procesos y sus principios y si es posible, incluso, preverlos.

Los griegos comenzaron a hacerse ya estas preguntas, intentando responderlas, y dando los primeros avances que conocemos hoy, como las teorías de Copérnico o Galileo sobre la astronomía y sus postulados. Con el paso del tiempo los recursos, tanto físicos como teóricos, fueron avanzando y es en 1671 cuando Newton acuña por primera vez el nombre de Ecuaciones Diferenciales Ordinarias, en su libro *Método de las fluxiones y series infinitas*[1], a las ecuaciones que relacionan una función con sus derivadas. Estas ecuaciones permitieron analizar y sustentar teóricamente procesos físicos como el problema de la cuerda vibrante, estudiado por científicos como d'Alembert o Euler, o el de la transferencia de calor, estudiado por Fourier en 1822.

Sin embargo, bien es sabido que la resolución de este tipo de ecuaciones no es nada fácil. La complejidad que alcanzan algunas de ellas para ser capaces de describir ciertos procesos físicos hacen que la obtención de soluciones exactas, para dichas ecuaciones, sea imposible. A pesar de ello las investigaciones no acabaron ahí, no dándose por vencidos, grandes matemáticos de la historia continuaron sus estudios y dieron lugar, así, a diversas técnicas que consistían en la aproximación numérica de estas, tan complejas, soluciones. Nacieron, de este modo, los métodos numéricos para la aproximación de ecuaciones diferenciales. Euler fue el primero en plantear una técnica de aproximación a la que le puso su propio nombre: el método de Euler, basado en la obtención de soluciones discretas a partir de la pendiente, dada por la derivada en cada punto, de un intervalo previamente fijado con una condición inicial dada.

Con el paso del tiempo las técnicas se fueron perfeccionando, dando lugar a un campo de estudio increíblemente amplio, y surgiendo, así, una gran variedad de métodos con propiedades muy diversas. Su finalidad era la de ser capaz de aproximar, de manera cada vez más exacta, la solución de los diferentes, y complejos, problemas que se planteaban. Además del perfeccionamiento de estas técnicas, también se produjeron avances en otros muchos ámbitos. En particular, a principios del siglo XX, aparecen las primeras ideas sobre la computación siendo en 1936 cuando Alan Turing introduce la "Máquina de Turing" en su trabajo *"On computable numbers, with*

[1] https://es.wikipedia.org/wiki/Ecuacion_diferencial

an application to the Entscheidungsproblem"[2], publicado por la Sociedad Matemática de Londres, a partir de la tesis de Church-Turing. Esto dio lugar a un tremendo avance en el campo de los métodos numéricos pues, los complejos cálculos que se requerían a mano para algunas de las técnicas eran, ahora, rápidos procedimientos que se realizaban a través de computadores. Además de la velocidad de cálculo y de la precisión que aportaban estos computadores a las aproximaciones anteriores, su estudio dio lugar a aspiraciones más ambiciosas: con papel y lápiz, la idea de un proceso físico se podía plasmar analíticamente, sin embargo, mediante un computador, la aproximación visual de dicho proceso estaba al alcance de la mano.

Comienzan así a darse los primeros avances en la simulación, y no sería hasta 1965 cuando Ivan Sutherland[3] da forma al concepto de "Realidad Virtual", al presentar su programa de investigación sobre el grafismo computerizado en la IFIP (Federación Internacional de Procesamiento de Información). En él propuso el modo de buscar que "las personas pudieran manipular e interaccionar directamente con el mundo virtual que se desarrollara, de manera que se pudiera conectar con dicho mundo de la misma forma que conectamos con el mundo real".

Dicha revolución continúa hasta nuestros días y ha tenido un repercusión asombrosa en la sociedad. Desde las primeras aplicaciones dedicadas al mundo de la ciencia ficción, hoy podemos encontrar usos de la realidad virtual en numerosos ámbitos:

- En eduación, se usa con fines prácticos para el diseño y modelado de todo tipo de estructuras ya sean físicas, arquitectónicas o sistemas del cuerpo humano, entre otros.
- En el mundo militar, podemos encontrar numerosos recursos destinados al entrenamiento de los cuerpos de élite en combate, permitiéndoles enfrentarse a situaciones de gran peligro en un entorno virtual seguro; así como en simulación de vuelo para las fuerzas aéreas, facilitando, y economizando, el aprendizaje de pilotos para un sinfín de situaciones.
- También en medicina ha jugado un papel importante, permitiendo la mejora en una gran variedad de campos, como en diagnósticos con modelos para la visualización de órganos vitales en 3D, la investigación de modelos sobre neurología o aplicaciones para la psicología y la cirugía.
- En arquitectura ha desempeñado un avance fundamental, otorgando la capacidad del estudio de hallazgos y ruinas reestructurándolos en tres dimensiones, o creando nuevas estructuras de forma digital con el fin de poder determinar, de forma previa, si es idónea su construcción en el entorno en el que vaya destinada a establecerse.
- Incluso el comercio ha sabido sacarle partido a sus cualidades haciendo uso de la poderosa herramienta que es Internet. A través de él hacen llegar hasta nuestros hogares, imágenes de los productos como calzados o prendas de vestir en un realismo que nos incita a su compra, facilitando el no ser necesario realizar ningún desplazamiento.
- Sin embargo, su mayor aplicación ha sido en el mundo de la industria del cine y los videojuegos, revolucionando ambos con el gran potencial que les ha aportado, siendo capaz de crear mundos inimaginables y de un realismo espectacular.

Este último punto ha sido el eje fundamental que ha motivado nuestro trabajo. Despertando la curiosidad que ya en el propio Grado nos fomentan con asignaturas como *Laboratorio de Modelización* o *Métodos Numéricos de las EDO* o *en Diferencias Parciales*, donde comenzamos a experimentar de forma práctica los conocimientos que hemos adquirido hasta el momento, nos

[2] https://es.wikipedia.org/wiki/Maquina_de_Turing
[3] https://es.wikipedia.org/wiki/Ivan_Sutherland

estimuló, de alguna forma, a seguir investigando para profundizar más sobre ello. A partir de la aplicación de métodos numéricos en estos entornos hemos buscado resaltar la importancia que éstos ejercen en las simulaciones y las aplicaciones de la Realidad Virtual.

Describiendo, de una forma global, el trabajo que presentamos, nos encontramos con dos capítulos altamente diferenciados.

En el primer capítulo nos centramos, de una forma más analítica y matemática, en el estudio de dos métodos numéricos que no nos es posible, por el tiempo limitado del que disponemos, desarrollar en nuestro grado: los métodos Adaptativos y los Simplécticos. Pretendemos conocer y comprender su estructura, su funcionalidad, y poder comprobarla en ejemplos sencillos que nos ayuden a verificar su utilidad.

En el segundo nos centramos de un modo más práctico en un problema concreto. A partir de un programa completamente funcional en el que podemos apreciar un primer ejemplo básico de aplicación de la realidad virtual, observamos un paño sustentado por dos de sus extremos que cuelga e interacciona con una esfera movida por el usuario. Dicha simulación se realiza mediante métodos numéricos y un entorno gráfico basado en el lenguaje C. Nuestro objetivo en este capítulo será, en un primer lugar, desenvolvernos con un nuevo lenguaje de programación, a partir de los conocimientos adquiridos en nuestro grado, para poder ser capaces de trabajar con él. En una segunda instancia, y una vez alcanzada cierta desenvoltura, analizar el código del programa con el que partimos, asumiéndolo como un problema práctico, el cuál hemos de analizar, entender, seccionar y, una vez comprendido, ver de qué manera somos capaces de resolverlo. En este caso particular compararemos la diferencia que resulta entre la aplicación del método original del programa, frente a otros estudiados en el grado, y mostraremos las posibles mejoras o defectos que resulten de la comparación, razonando de este modo, a qué son debidas. Asimismo, añadiremos una mejora a la simulación, confiriéndola de una manera más física, aportando la ley de Hooke para simular el comportamiento del paño. Para llegar a dicho punto, pasaremos por el análisis del entorno gráfico con el que trabaja el programa, OpenGL, de uso realmente extendido en el ámbito de la simulación con gráficos a nivel computacional.

Remarcar que en la elaboración de estos capítulos nos hemos vistos sujetos y limitados a la capacidad de nuestra máquina de trabajo que, en ocasiones, no ha sido capaz de soportar las computaciones y renderizaciones que llevábamos a cabo. Además, el entorno de trabajo sobre el que se ha elaborado el programa, Visual Studio, presentaba diversos problemas de compatibilidad con algunas de las librerías de trabajo. Sin embargo, gracias a la ayuda tanto de mi tutor Eliseo, como del profesor Gregorio Martínez, fuimos solventando este tipo de problemas, más informáticos, dando lugar a un proyecto del que hemos acabado sintiéndonos realmente orgullosos. Los resultados visuales que hemos conseguido alcanzar con este trabajo y el juego que nos ha ofrecido su dinamización, nos han llevado a sentirnos verdaderamente satisfechos.

De este modo pretendemos mostrar, de una forma básica, las aplicaciones prácticas que pueden llegar a tener las matemáticas en los entornos virtuales intentando tomar conciencia, al mismo tiempo, del gran avance que se ha conseguido desarrollar con el paso del tiempo, y hasta nuestros días.

Introduction

"Les mathématiques sont l'alphabet avec lequel Dieu a écrit l'universe."
— Galileo Galilei.

Les mathématiques dominent la vie. Après quatre années d'études, n'importe quel étudiant de mathématiques connaît la transcendance que celles-ci présentent dans le monde. L'univers, et tout ce qui nous entoure, est dirigé par des lois physiques qui sont dominées par les premières, qui soutiennent sa base. Tout processus physique dont on peut être conscients chaque jour a un principe mathématique derrière. Si ce n'est pas ansi, on peut en arriver à se demander à quel point on est capable de conaître ces processus et leurs principes et, même si c'est possible, le prévoir.

Les Grecs ont comencé se poser déjà ces questions, essayaient de les répondre en réalisant les premièrs progrès qu'on conaît aujourd'hui, comme les théories de Copernic ou Galilée vers l'astronomie, et ses postulés. Au fil du temps, les ressources, a la fois physiques aussi bien que théoriques, ont fait des progrès; et c'est en 1671 quand Newton a fixé pour la première fois le nom d'Équation Differentielle Ordinaire dans son livre *Méthodes de les fluxions et les séries infinites*, aux équations qui mettent en rapport une fonction avec ses dérivées. Ces équations permettent d'analyser et soutenir théoriquement des processus physiques comme le problème de la corde vibrante, étudié par des scientifiques comme d'Alembert ou Euler, ou ceux de la transfer de la chaleur étudié par Fourier en 1822.

Néanmoins, c'est un fait bien connu que la résolution de ce tipe d'équation n'est pas du tout facile. La complexité qu'atteignent quelques unes d'entre elles, pour être capables de décrire certains processus physiques, font que l'obtention des solutions exactes, pour ces équations, soit impossible. Par contre, les recherches n'abiytussent pas ici, car sans se déclarer vaincu, des grands mathématiciens de l'histoir ont continué leur études et ils ont donné lieu, ainsi, aux diverses techniques qui consistent à l'approximation numérique de ces solutions aussi complexes. Sont nées ainsi les méthodes numériques pour l'approximation d'équations différentielles ordinaires. Euler a été le premier qui a proposé une technique d'approximation qui porte son propre nom: la méthode d'Euler, basé sur l'obtntion de solutions discrètes à partir de la pente donné pour la dérivée en chaque point, d'un intervalle au préable fixé avec une condition initiale donnée.

Au fil du temps, les techniques se sont perfectionées donnant leiu au champ d'étude incroyablement approfondi ce qui a produit ainsi une grande variété des méthodes présentant des propriétés très diverses. Sa finalité était celle d'être capable d'approcher, d'une manière de plus en plus exacte, la solution des différents et complexes problèms qui se sont posés. En plus du perfectionnement de ces techniques, des progrès dans plusieurs autres cadres se sont aussi produit. En particulière, au debut du XXe siècle son apparues les premières idées sur la computationet c'est en 1936 qu'Alan Turing introduit la "Machine de Turin" dan son travail *"On computable*

numbers, with an application to the Entscheidungsproblem", publié par la Société Mathématique de Londres, a partir de la thèse de Church-Turin. Ce qui a donné lieu à un énorme progrès dans le champ des méthodes numériques puisque les complexes calculs qui nécessitaient d'être fait à la main pour certaines de ces techniques devenaient, maintenant, des procédures rapides qu'on se realisait à travers des ordinateurs. En plus de la vitesse de calcul et de la précision qu'apportent ces ordinateurs aux précisions précédentes, son étude a donné lieu à des aspirations plus ambitieuses: avec un papier et un crayon, l'idée d'un processus physique pouvait s'exprimer d'une façon anatlytique; cependent, grâce à un ordinateur, l'approximation visuelle de ce processus était à portée de la main.

Les premiers processus en matière de simulation commencent à se concrétiser et ce en sera qu'en 1965 quand Ivan Sutherland donne forme au concept de "Réalité Virtuelle" en présentant son programme d'investigation su le graphisme informatisé dans la IFIP (Fédération Internationelle de Traitement de l'Information) dans lequel il proposse le mode de chercher que les personnes pussent manipuler et interagir avec le monde virtuelle de façon qu'on puisse connecter avec ce monde de la même façon qu'on connecte avec le monde réel.

Cette révolution continue jusqu'à nos jours avec une répercussion étonnante dans la société. Depuis les premières applications dédiées au monde de la science fiction, aujourd'hui on peut trouver des usages de la réalité virtuelle dans de nombreux domaines:

- Dans l'éducation on l'utilise avec des finalités practiques pour le dessin et le modelage de toutes sortes de structures aussi bien physiques, architectonique ou systèmes du corps humain, entre autres.
- Dans le monde militaire, on peut trouver de nombreuses ressources destinées à l'entraînement des corps d'élite en combat, en permettrant de cette façon se confronter à des situations vraiment dangereuses dans un environnement sûr; ainsi que dans des simulations pour les forces aériennes,facilitant tout en économisant l'apprentissage des pilotes pour une infinité des sitiations.
- Aussi en médecine, ella a joué un rôle très important, elle a permis l'amélioration d'une grande variété de domaines, comme pour les diagnostiques à modèles pour la visualisation d'orgues vitaux en 3D, la recherche de modèles en neurologie ou des applications pour la psicologie et la chirugie.
- En architecture, elle a joué un rôle fondamental en conférant la capacité d'étude de découvertes et des ruines en les restructurant 3D ou en créant des nouvelles structures d'une façon digital afin de pouvior déterminer, d'une manière préable, si sa construction est recommandable à l'endroit dans lequel elles iraient destinées.
- Même le commerce a su le tirer parti en faisant ussage du puissant outil qui est Internet en mettant à portée de nos mains des images de produits comme des chaussures ou des vêtements avec un réalisme qui nous incite à leur achat sans avoir à réaliser aucun déplacement.
- Néanmoins, sa plus grande application s'est produit dans le monde de l'industrie du cinéma et les jeux vidéo en révolutionnant tout les deux grâce au grand potentiel présenté tout en étant capable de créer des mondes inimaginables et d'un réalisme spectaculaire.

Ce denière point a été l'axe fondamental qui a motivé notre travail. Il a éveillé la cuiriosité, qui déjá pendant les études nous a encouragé avec des matières comme *Laboratoire de Modélisation*, où on commence expérimenter, d'une forme practique, les connaissance qu'on a acquises jusqu'à

maintenant; d'une certaine façon nous avons été stimulé à continuer avec nos recherches pour approfondir davantage. A partir de l'application des méthodes numériques dans ces endroits, nous avons cherché à souligner l'importance que celles-ci exercent dans les simulations et les applications de la Réalité Virtuelle.

En decrivant, d'une façon globale, le travail qu'on presente, on distingue deux chapitres hautement différenciés:

Dans le premier chapitre on se concentre, d'une forme plus analitique et mathématique, sur l'étude de deux méthodes numériques qu'on s'est pas possible, pour le temps limité duquel on dispose, développer pendant les études: les méthodes Adaptatives et les Symplectiques. On prétend connaître et comprendre leur structure, leur fonctionalité, et pouvoir les vérifier avec des exemples simples qui nous aident à avérer son utilité.

Dans le deuxième chapitre, on se centre, d'une forme plus practique, sur un problème concret. À partir d'un programme complètement fonctionnel, dans lequel on peut apprécier un premier exemple basique d'application de la réalité virtuelle, on observe un drap soutenu par deux de ses extrêmes qui pend et interagit avec une sphère mué par l'utilisateur. Cette simulation se réalise grâce à des méthodes numériques et un environement graphique basé sur le langage C. Notre but, dans ce chapitre sera, dans un premier lieu, appliquer un nouveau langage de programmation, a partir des connaisances acquises dans notre dégrée pour pouvoir être capables de travailler avec lui. Dans un deuxième temps, et une fois atteinte une certaine désinvolture, analyser le code du programme avec lequel on a démaré en l'assumant comme un problème practique qu'on doit analyser, comprendre, sectionner et, un fois compris, voir de quelle façon on est capable de le résoudre. Dans ce cas particulière, on comparera la différance qui résulte entre l'application de la méthode original du programme, face à d'autres étudiés dans le dégrée et on montrera les possibles améliorations ou défauts qui résultent de la comparaison, en justifiant, de cette manière, a quoi elles sont dues. De même, on ajutera une amélioration à la simuation, en la conférant d'une forme plus physique, en apportant la Loi de Hooke pour simuler le comportement du drap. Pour arriver a ce point, on passera pour l'analyse de l'endroit graphique avec lequel travaille le programme, OpenGL, d'un ussage réalement extendu dans le cadre de la simulation avec des graphiques au niveau informatique.

Remarquer que, dans l'élaboration de ce chapitre, nous avons été exposés et limité à la capacité de notre machine de travail qui, à centains moments, n'a pas été capable de supporter les computations et les renderisations qu'on réalisait. En plus, l'environnement de travail sur lequel on a élaboré le programme, Visual Studio, presentait des différents problèmes de compatibilité avec certaines librairies de travail. Tout de même, grâce à l'aide aussi bien de mon tuteur comme du professeur Gregorio Martínez, nous avons pu ce tipe de problèmes, surtout de type informatiques, donnant lieu à un project dont on a fini par se sentir vraiment fiers.

De cette façon on prétend montrer, d'une forme basique, les applications practiques qui peuvent avoir les mathématiques dans les environnements virtuels, en esayant d'être conscients, au même temps, du grand progrès atteint au fil du temps et jusqu'à nos jours.

Capítulo 1

Métodos Numéricos

"Todo saber tiene de ciencia,
lo que tiene de matemática."
— Poincaré.

La física, una de las ramas más importantes de la ciencia, sabemos que es capaz de modelar y explicar, de forma razonada y científica, todos los fenómenos que nos rodean y acompañan cada día: procesos atmosféricos, cambios de temperatura, gravedad y un sinfín de ejemplos más. Dicho modelado tiene una base fundamental y viene regido por ecuaciones matemáticas, las cuales sustentan y dan consistencia a esta teoría. Dichas ecuaciones se presentan como Ecuaciones Diferenciales Ordinarias (EDO) o Ecuaciones en Derivadas Parciales (EDP), dos campos que abarcan un impresionante mundo de aplicaciones reales, a partir de análisis complejos de problemas físicos y matemáticos.

En este capítulo nos centraremos únicamente en lo relacionado con las Ecuaciones Diferenciales Ordinarias y nos basaremos, fundamentalmente, en las referencias [13] y [8]. Bien es sabido por los estudiantes de matemáticas que aquellos problemas y ecuaciones que tienen una solución exacta, la cual podemos calcular de forma explícita, son, si bien, escasos, y las EDO no lo iban a ser menos. Calcular explícitamente la solución de una EDO puede ser, en muchas ocasiones, una ardua tarea y ello en el caso de que se pueda. Debido a la imposibilidad de poder dar dichas soluciones, se han desarrollado numerosas técnicas y resultados teóricos que nos dan aproximaciones e intentan conocer al máximo las propiedades y características de estas soluciones. Es así como surgen los métodos numéricos.

En nuestro grado en matemáticas destacan tres asignaturas enfocadas directamente a este tema: *Métodos Numéricos de las Ecuaciones Diferenciales Ordinarias*, *Métodos Numéricos de las Ecuaciones en Derivadas Parciales* o la *Teoría Cualitativa de las EDO*, esta última enfocada directamente, no sobre los resultados explícitos, sino sobre las propiedades de éstos. Como objeto de nuestro estudio tomaremos como referencia, por lo antes indicado, la primera de ellas.

Si bien es cierto que el campo de estudio es tremendamente amplio, nosotros nos centraremos en la aproximación de soluciones de las Ecuaciones Diferenciales mediante métodos numéricos, en particular, dos técnicas que no se tratan, propiamente, en el grado: Los métodos Adaptativos y los Simplécticos.

De esta manera desarrollaremos algunas competencias del grado[1] como: *"CGM-1. Comprender y utilizar el lenguaje matemático, CGM-2. Conocer demostraciones rigurosas de algunos teoremas clásicos'", "CGM-3. Asimilar la definición de un nuevo objeto matemático, en términos de otros ya conocidos, y ser capaz de utilizarlo en otros contextos", "CGM-5. La capacidad del aprendizaje autónomo de nuevos conocimientos y técnicas", "CGM-6. Resolver problemas de matemáticas mediante habilidades de cálculo básico y otras técnicas".*

1.1. Motivación

Como bien sabemos, los métodos numéricos se basan en la realización de un algoritmo que especifica una secuencia de operaciones algebraicas que generan la solución del problema tratado. Dicho procedimiento vio la luz con Euler, quien desarrolló el primer método entorno a 1770, método que lleva su propio nombre, para la aproximación de dichas soluciones. Dicha idea era simple, aunque no sencilla: aproximar una solución continua dada en un intervalo, $[0, T]$, por medio de la discretización de éste en N puntos, partiendo de una condición inicial y avanzando, con un paso de tiempo $h = T/N$, en la dirección de su pendiente proporcionada por la derivada en dicho punto. Este tipo de problemas reciben el nombre de Problemas de Valor Inicial (PVI). Nosotros nos centraremos, en concreto, en la resolución de Problemas de Cauchy, caracterizado por ser de la forma:

$$\begin{cases} y'(t) = f\left(t, y(t)\right) & t \in [0, T] \\ y(t_0) = y_0 \end{cases} \tag{1.1}$$

donde $y(t)$ es la variable dependiente, t es la variable independiente que se desplaza en el intervalo $[0, T]$, e, y_0 es el valor inicial dado por hipótesis.

Dentro de los métodos numéricos podemos elaborar una distinción, algo global, de ellos: Métodos de un paso y Métodos multipaso.

- Por un lado podemos encontrar los métodos de un paso. Estos métodos calculan predicciones futuras, y_{n+1}, a partir de la información aportada por el punto inmediatamente anterior, sin otorgar mayor importancia al resto de los ya calculados. Esto tiene la ventaja de no arrastrar los posibles errores que se hayan ido cometiendo a lo largo de sus aproximaciones, sin embargo, también tienen el inconveniente de que pueden perder la linealidad para salvaguardar el orden. Basándonos en la notación proporcionada por [11], para un problema de Cauchy, dichos métodos vienen determinados por la fórmula general:

  $$y_{n+1} = y_n + h\phi_f(t_n, y_n, y_{n+1}; h) \tag{1.2}$$

  Donde ϕ_f es la función incremento, la cual, dictamina si el método es explícito, caso de que ϕ_f no dependa de y_{n+1}, o si el método es implícito, en el caso contrario. La consistencia en estos métodos implica la convergencia de los mismos, dado que se obtiene la estabilidad por construcción.

- Por otro lado tenemos los métodos multipaso. Estos métodos toman la información de puntos anteriores ya calculados con el fin de construir un polinomio de interpolación con dichos valores, incluyendo el actual en caso de ser implícito. De esta forma, la curvatura de las líneas conectan las aportaciones de estos valores previos sobre la trayectoria de la solución. A diferencia de los métodos de un paso, éstos mantienen siempre la linealidad en los cálculos de sus diversas iteraciones, pero requieren de la consistencia y la estabilidad a

[1] http://www.um.es/web/matematicas/contenido/estudios/grados/matematicas/descripcion

cero para asegurar la convergencia. De este modo, para calcular la aproximación y_{n+k}, a partir de sus resultados anteriores $y_0, \cdots, y_{k-1}$, la fórmula general que siguen estos métodos viene dada por

$$y_{n+k} = -\sum_{j=0}^{k-1} a_j y_{n+j} + \phi_f(t_n, y_n, y_{n+1}, \cdots, y_{n+k})$$

En este caso, $\phi_f = \sum_{j=0}^{k} b_j f(t_{n+j}, y(t_{n+j}))$ y, además, se pide que $a_0^2 + b_0^2 \neq 0$. Que el método sea explícito o implícito viene determinado por el valor de b_k, que indicará si el método es de primer tipo (con $b_k = 0$), o del segundo (en el otro caso).

1.2. Métodos de un paso: Adaptativo y Simpléctico

Dentro de la resolución de Ecuaciones Diferenciales Ordinarias a través de métodos numéricos existe una gran diversidad de procedimientos. Nosotros nos centraremos en el estudio de los casos para los métodos de un paso. Dado un PVI, la forma más sencilla de avanzar en la aproximación de soluciones, a partir de su condición inicial, es la proporcionada por el método de Euler Explícito, mencionado al principio de esta sección y que presenta la siguiente estructura

$$\begin{cases} u_{n+1} = u_n + hf(t_n, u_n) \\ u_0 = y_0 \end{cases} \tag{1.3}$$

donde y_0 es la condición inicial, h el paso de tiempo determinado previamente por la partición (como el cociente T/N) y $t_n = hn$ variando n entre 0 y N. Sin embargo, este método presenta la desventaja de ser un método de primer orden, es decir, el error de aproximación con respecto a la solución exacta de un problema es de la magnitud de $O(h^2) \approx Kh^2$, con K una constante. La demostración de este resultado podemos verla a continuación basándonos en el resultado de [10, pág 27]

Teorema 1.2.1. *El método de Euler Explícito converge con orden 1, para una función $f \in \mathcal{C}^1$ de tipo Lipschitz.*

Demostración. Dado un problema de Cauchy del tipo (1.1), supongamos $f \in \mathcal{C}^1$ al menos. Para la aproximación del punto $y(t_{n+1})$ podemos recurrir al desarrollo de Taylor obteniendo

$$y(t_{n+1}) = y(t_n) + hy'(t_n) + \frac{1}{2}h^2 y''(\xi_n) \quad \text{con} \quad t_n < \xi_n < t_{n+1}$$

es decir,

$$y(t_{n+1}) = y(t_n) + hf(t_n, y(t_n)) + \frac{1}{2}h^2 y''(\xi_n) \quad \text{con} \quad t_n < \xi_n < t_{n+1}$$

De este modo, en cada iteración, tenemos

$$\begin{aligned} e_{n+1} := y(t_{n+1}) - y_{n+1} &= \Big(y(t_n) + hf(t_n, y(t_n)) + \frac{1}{2}h^2 y''(\xi_n)\Big) - \Big(y_n + hf(t_n, y(t_n))\Big) = \\ &= e_n + h\Big(f(t_n, y(t_n)) - f(t_n, y(t_n))\Big) + \frac{1}{2}h^2 y''(\xi_n) \end{aligned}$$

Usando ahora la propiedad de Lipschitz sobre f y tomando $M = \text{máx}_{t_0<t<t_0+T} h^2 \frac{|y''(t)|}{2}$ podemos acotar el error como:

$$e_{n+1} \leq (1 + hL)|e_n| + M$$

De donde, de forma recursiva, se obtiene

$$\begin{aligned} |e_1| &\leq (1+hL)|e_0| + M \\ |e_2| &\leq (1+hL)|e_1| + M \leq (1+hL)^2|e_0| + \Big((1+hL)+1\Big)M \\ &\vdots \\ |e_{n+1}| &\leq (1+hL)^{n+1}|e_0| + \Big(1+(1+hL)+(1+hL)^2+\ldots+(1+hL)^n\Big)M \end{aligned}$$

y, haciendo uso de la suma geométrica y de la desigualdad $(1+hL)^{n+1} \leq e^{TL}$ llegamos a

$$|e_{n+1}| \leq (1+hL)^{n+1}|e_0| + \frac{(1+hL)^{n+1}-1}{(1+hL)-1}M \leq e^{TL}|e_0| + \frac{e^{TL}-1}{2L}\text{máx}_{t_0<t<t_0+T}|y''(t)|h$$

De forma que nuestro error en el paso $n+1$ depende del error inicial cometido, e_0, y el paso h establecido. Puesto que, generalmente, el valor de h es fijo, podemos describir el error en términos de ella de forma que $|e_0| = C_0 h$ y, así, concluir que

$$|e_n| \leq Ch, \quad 0 \leq n \leq N = T/h$$

con C constante, es decir,

$$e_n = O(h), \ h \longrightarrow 0^+$$

y que, por tanto, el método converge a cero con orden 1 respecto a h. □

Esta demostración nos ayuda a comprender un poco mejor cómo actúan los errores globales a lo largo de las iteraciones de un método numérico, en este caso el de Euler Explícito. De esta forma podemos dar una estimación del error local que será del orden de una $O(h^2)$. Como observamos al acotar este valor por la constante M de la prueba anterior, y teniendo en cuenta que tomamos un número N de pasos para dicha acotación, esto provoca una contribución final al error global de la forma $O(h)$, como bien se deduce al final de la prueba.

Además, esta prueba es válida para un método general del tipo (1.2) usando

$$l(t;h) = y(t+h) - y(t) - h\phi_f(t, y(t), y(t+h); h) + O(h^{p+1})$$

y que

$$||\phi_f(t,y,z;h) - \phi_f(t,\tilde{y},\tilde{z};h)|| \leq L_\phi\left(|y-\tilde{y}| + |z-\tilde{z}|\right)$$

donde L_ϕ es la constante de Lipschitz asociada al método ϕ_f. En [11] podemos encontrar un argumento más extendido sobre este resultado.

Es por ello que, con el fin de disminuir estos errores y aumentar el orden de convergencia a cero, a lo largo del tiempo se han ido desarrollando métodos y técnicas más elaboradas. Siendo así se encuentran, entre otros, los métodos de la familia Runge-Kutta, los cuales serán de vital importancia, sobre todo, en el segundo capítulo de nuestro trabajo. Dichos métodos consiguen tener órdenes de convergencia mayor, a costa de perder la linealidad para mantenerlo. En lugar de calcular la aproximación y_{n+1} a partir de la pendiente obtenida por el punto y_n, toman s muestras de pendientes (según las etapas del método considerado), correspondientes a aproximaciones intermedias entre ambos puntos a fin de realizar un promedio de ellas para calcular la aproximación definitiva y_{n+1} mejorándola, de esta forma, a la de otros métodos de órdenes más bajos. Así, el esquema de resolución dependerá del número de etapas del método que

afectará a las combinaciones lineales del cálculo de las pendientes empleadas en la aproximación. Un esquema general del método Runge-Kutta de s etapas es el siguiente:

$$\begin{aligned}
u_{n+1} &= u_n + \sum_{i=1}^{s} \beta_i k_i \\
&\text{donde} \\
k_1 &= f(t_n + c_1 h, u_n + h \sum_{j=1}^{s} \alpha_{1j} k_j) \\
k_2 &= f(t_n + c_2 h, u_n + h \sum_{j=1}^{s} \alpha_{2j} k_j) \\
k_3 &= f(t_n + c_3 h, u_n + h \sum_{j=1}^{s} \alpha_{3j} k_j) \\
&\vdots \\
k_s &= f(t_n + c_s h, u_n + h \sum_{j=1}^{s} \alpha_{sj} k_j)
\end{aligned}$$

De igual modo que en el esquema de Euler, h hace referencia al paso fijado al inicio, y las constantes α_{ij}, β_i y c_i vienen determinadas para cada método por un "Tablero de Butcher"[2]. El tablero de Butcher viene determinado por la siguiente estructura:

$$\begin{array}{c|ccccc}
c_1 & \alpha_{11} & \dots & & & \alpha_{1s} \\
c_2 & \alpha_{21} & \alpha_{22} & \dots & & \alpha_{2s} \\
c_3 & \alpha_{31} & \alpha_{32} & \ddots & & \\
\vdots & \vdots & \vdots & \ddots & & \vdots \\
c_s & \alpha_{s1} & \alpha_{s2} & \dots & \alpha_{ss-1} & \alpha_{ss} \\
\hline
 & \beta_1 & \beta_2 & \dots & \beta_{s-1} & \beta_s
\end{array}$$

el cual, además, debe verificar las siguientes restricciones:

- $\sum_{i=1}^{s} \beta_i = 1$
- $c_i = \sum_{j=1}^{i-1} \alpha_{ij}$

Cabe remarcar que un tablero así definido determina, genéricamente, un método de Runge-Kutta implícito. Si, por el contrario, el tablero de Butcher es triangular inferior con la diagonal idénticamente nula, entonces nos encontraremos en el caso de los métodos Runge-Kutta explícito,

[2]Introducido por John Butcher en los 60 para la representación de los métodos de Runge-Kutta a partir de las series de Taylor de las soluciones ([7])

que serán los objetos de nuestro estudio. Éstos, más simples, vendrán definidos como sigue:

$$\begin{aligned} u_{n+1} &= u_n + \sum_{i=1}^{s} \beta_i k_i \qquad (1.4) \\ &\text{donde} \\ k_1 &= f(t_n, u_n) \\ k_2 &= f(t_n + c_2 h, u_n + h\alpha_{21} k_1) \\ k_3 &= f(t_n + c_3 h, u_n + h(\alpha_{31} k_1 + \alpha_{32} k_2) \\ &\vdots \\ k_s &= f(t_n + c_s h, u_n + h \sum_{j=1}^{s-1} \alpha_{sj} k_j) \end{aligned}$$

Con todo y con ello, este tipo de métodos no dejan de tener una dependencia directa del paso h fijado inicialmente, por lo que su elección puede condicionar enormemente el funcionamiento del propio método. De modo que, cómo determinar cuál es el paso de tiempo h más indicado en según qué caso, puede ser realmente interesante. Con el fin de solventar este problema, comienzan a desarrollarse técnicas para controlar esta "dependencia del paso de tiempo" de los métodos, nacen así los Métodos de Paso Adaptativo. Asimismo, y con el fin de mantener la estabilidad durante largos periodos de computación, aún con la dependencia del paso citada, se desarrollaron otras técnicas que buscarán la conservación de la energía del sistema, dando lugar con ello, a los métodos conservativos y, más específicamente, los Simplécticos.

1.2.1. Métodos Simplécticos

En esta sección nos dedicaremos a estudiar un caso particular de métodos Conservativos. Como bien hemos indicado en la motivación de este capítulo: Las Ecuaciones Diferenciales nos sirven para aproximar procesos físicos de la realidad cotidiana, procesos físicos basados en la energía de los sistemas que los sustentan. Siendo así, parece claro que una de las características más importante que hemos de buscar en los métodos que nos ayuden a aproximar dichos procesos sería la conservación de esta energía. Sin embargo, encontrar métodos con estas características no es una tarea fácil, por ello, dicha "conservación de la energía" se ve sustituida por la conservación de la simplecticidad, cualidad mucho más sencilla de encontrar en métodos numéricos de aproximación.

La palabra "simpléctico" hace referencia a la conservación de áreas, volúmenes u orientaciones, es decir, a la conservación de la energía de los sistemas que los producen de un modo más geométrico. Es por ello que los métodos que poseen esta propiedad son de gran interés en el estudio de la resolución de ecuaciones mediante métodos numéricos, ya que, dicha conservación tenderá a proporcionar una mejor aproximación a la solución exacta que los métodos tradicionales, en según qué casos.

En lo referente a estos métodos nos ha sido de gran utilidad consultar libros como Hairer [11] o Holmes [13] sobre ecuaciones y sistemas diferenciales. Haciendo alusión al primero, sabemos que el estudio de estos métodos comenzó entorno a 1988, tras algunos estudios pioneros anteriores (para mayor información se puede consultar la página 312 del mismo). El segundo nos remarca la importancia de los métodos simplécticos: una de la primeras razones es que muchos de estos problemas son, de por sí, simplécticos, luego simplemente requerimos preservar una propiedad con el método que ya tiene el propio problema. Otra razón es, como ya hemos indicado un par de párrafos más arriba, que es mucho más sencillo encontrar métodos numéricos que conserven la

simplecticidad que aquellos que conserven la energía. Y la tercera, y más importante, es que los métodos simplécticos son capaces de, aún no conservándolo, obtener resultados muy cercanos a la solución exacta del Hamiloniano asociado al sistema, cualidad necesaria para la conservación de la energía.

El Hamiltoniano hace referencia a la transformada de Legendre de la función de Euler-Lagrange, también definida como lagrangiano, que viene dada por la ecuación

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{y}}\right) - \frac{\partial L}{\partial y} = 0$$

para el funcional

$$\Phi = \int_{t_0}^{t_1} L(y, \dot{y}, t)dt$$

con $L = T - U$, diferencia entre energía cinética y potencial para un sistema de masas $(q_1, \ldots, q_n)$ dado.

Además, se puede probar que las coordenadas de n puntos de masa, en el espacio de configuración del sistema, están sujetas a dichas ecuaciones de Euler-Lagrange. Por otro lado, un sistema de ecuaciones lagrangiano es equivalente, a partir de una transformada de Legendre, a un sistema de primer orden

$$\begin{cases} \dot{p} = -\frac{\partial H}{\partial q} \\ \dot{q} = \frac{\partial H}{\partial q} \end{cases}$$

donde $p = \partial L/\partial \dot{q}$, y $H(p, q, t) = p\dot{q} - L(q, \dot{q}, t)$, el cual, recibe el nombre de Hamiltoniano.

Bajo estos supuesto, el hamiltoniano H es la energía total de un sistema dado, tomando $H = T + U$, suma de la energía cinética y la potencial. La prueba de estos resultados no se ha hecho debido a que todos estos conceptos son transversales al objetivo perseguido en este trabajo, y la extensión de su desarrollo excedía lo convenido en este sentido. Sin embargo, todos ellos se pueden encontrar desarrollados y explicados en el capítulo 3 de Mecánica Clásica [4].

Características

Dada una ecuación o sistema de ecuaciones diferenciales ordinarias, basadas en la mecánica newtoniana, donde una fuerza es aplicada sobre el sistema, podemos considerar su Hamiltoniano. Dicho hamiltoniano se descompone como suma de la energía cinética y la potencial del sistema, de este modo podemos observar fácilmente cómo éste es constante ([13, pág 26], [11, pág 313]), es decir, la energía se conserva. Dicho esto, podemos dar una caracterización de los métodos simplécticos a partir del siguiente enunciado obtenido de [13, pág 30]:

Teorema 1.2.2. *Supongamos que*

$$\begin{cases} y_{j+1} = f(y_j, v_j) \\ v_{j+1} = g(y_j, v_j) \end{cases}$$

es una aproximación por diferencias finitas del sistema

$$\begin{cases} y' = v \\ v' = \frac{1}{m}F(y) \end{cases}$$

dada por un cierto método numérico. Entonces diremos que el método es simpléctico si y sólo si $f_y g_v - f_v g_y = 1 \ \forall y, v$*; o lo que es lo mismo,*

$$det \begin{pmatrix} f_y & f_v \\ g_y & g_v \end{pmatrix} = 1$$

Más gráficamente, si suponemos que tenemos tres condiciones iniciales $\mathrm{y}_a = (y_a, v_a)$, $\mathrm{y}_b = (y_b, v_b)$ e $\mathrm{y}_c = (y_c, v_c)$ tales que, tras un paso de tiempo h, un método produce los valores $\mathrm{y}_A = (y_A, v_A)$, $\mathrm{y}_B = (y_B, v_B)$ e $\mathrm{y}_C = (y_C, v_C)$, respectivamente. Entonces, la propiedad que estamos buscando resulta al observar las diferencias entre los paralelogramos obtenidos de los tres primeros valores, y su trasformación a los tres segundos, **Figura 1.1**. De esta forma, asumiendo que y_b e y_c están cerca de y_a con una distancia h pequeña, entonces, si ambos paralelogramos tienen el mismo área en el orden de h^2, y la misma orientación, se dirá que dicho método es simpléctico.

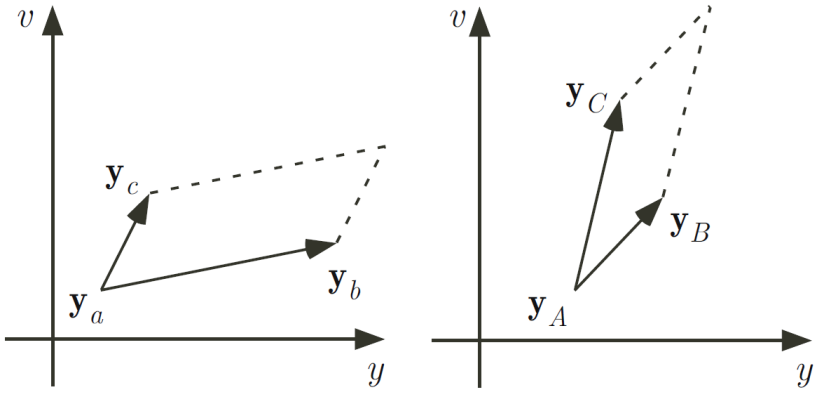


Figura 1.1: Plano de fase de los paralelogramos resultantes de la aproximación simpléctica usando el método, en un paso de tiempo, tal que $a \to A$, $b \to B$ y $c \to C$. Figura obtenida de [13, pág 30]

Velocity Verlet

Basándonos en el ejemplo de [13, pág 27], y partiendo del estudio de una ecuación regida por la segunda ley de Newton, sin dependencia de la velocidad, se obtiene

$$\frac{d^2y}{dt^2} = \frac{1}{m}F(y) \tag{1.5}$$

Dicha ecuación podemos convertirla en un sistema de primer orden añadiendo, simplemente, $y' = v$. De esta forma, aplicando la regla del trapecio, queda

$$\begin{cases} y_{j+1} = y_j + \frac{h}{2}(v_{j+1} + v_j) \\ v_{j+1} = v_j + \frac{h}{2m}(F(y_{j+1}) + F(y_j)) \end{cases}$$

Sin embargo, esto produce un método implícito, que puede suponer una carga computacional importante. No obstante, podemos explicitarlo aproximando $v_{j+1} = v_j + \frac{h}{m}F(y_j)$, de la primera ecuación, mediante el método de Euler. Quedando así

$$\begin{cases} y_{j+1} = y_j + hv_j + \frac{h^2}{2m}F(y_j) \\ v_{j+1} = v_j + \frac{h}{2m}(F(y_{j+1}) + F(y_j)) \end{cases} \tag{1.6}$$

Comprobar que, efectivamente, el método de Verlet es simpléctico no es difícil, bastará aplicar el Teorema 1.2.2. Dado el sistema (1.6), podemos definir las funciones f y g del teorema como

$$\begin{cases} f(y,v) = y + hv + \frac{h^2}{2}F(y) \\ g(y,v) = v + \frac{h}{2}\left[F\left(y + hv + \frac{h^2}{2}F(y)\right) + F(y)\right] \end{cases}$$

De donde

$$\begin{cases} f_y(y,v) = 1 + \frac{h^2}{2}F'(y) \\ f_v(y,v) = h \end{cases} \qquad \begin{cases} g_y(y,v) = \frac{h}{2}F'\left(y + hv + \frac{h^2}{2}F'(y)\right)\left(1 + \frac{h^2}{2}F'(y)\right) + \frac{h}{2}F'(y) \\ g_v(y,v) = 1 + \frac{h^2}{2}F'\left(y + hv + \frac{h^2}{2}F'(y)\right) \end{cases}$$

Y haciendo el correspondiente determinante queda

$$\begin{aligned} f_y g_v - f_v g_y &= \left[1 + \frac{h^2}{2}F'\left(\cancel{y + hv + \frac{h^2}{2}F(y)}\right)\right]\left(1 + \frac{h^2}{2}F'(y)\right) - \\ &- \frac{h^2}{2}F'\left(\cancel{y + hv + \frac{h^2}{2}F(y)}\right)\left(1 + \frac{h^2}{2}F'(y)\right) + \frac{h^2}{2}F'(y) = 1 + \frac{h^2}{2}F'(y) - \frac{h^2}{2}F'(y) = \boxed{1} \end{aligned}$$

Es decir nuestro método es simpléctico.

Consideremos ahora una fuerza $F(y) = -y$ y una masa $m = 1$ de forma que (1.6) queda

$$\begin{cases} y_{j+1} = y_j + hv_j - \frac{h^2}{2m}y_j \\ v_{j+1} = v_j - \frac{h}{2m}(y_{j+1} + y_j) \end{cases}$$

su hamiltoniano, $H = \frac{1}{2}(y')^2 + \frac{1}{2}y^2$, y las condiciones iniciales dadas por $y(0) = 1$ e $y'(0) = 0$. En esta situación, la solución exacta viene determinada por

$$\begin{cases} y(t) = cos(t) \\ v(t) = -sin(t) \end{cases}$$

con un hamiltoniano constante $H(t) = \frac{1}{2}$, es decir, la solución se sitúa en el círculo unidad: $y^2 + v^2 = 1$. Resolviendo este sistema mediante MATLAB para $T = 4000$ y $h = 0.5$, obtenemos los resultados de las figuras: **Figura 1.2**, **Figura 1.3** y **Figura 1.4**.

Donde queda evidenciado que, RK4, a pesar de ser un método de orden superior al método del Velocity Verlet (orden cuatro frente a orden dos), éste pierde energía a lo largo del tiempo, lo que hará que, finalmente, deje de aproximar bien a la solución exacta. Es por ello que este tipo de métodos simplécticos, a pesar de tener un orden inferior a otros métodos de un paso, se utilicen en aproximaciones o simulaciones de largos periodos de tiempo ya que, en cierto modo, aseguran la consistencia del sistema.

Por otro lado, y apoyándonos en el ejercicio 1.22 de [13, pág 39], dado el método Velocity Verlet (1.6) dependiente de la velocidad, éste es equivalente al método de Verlet, el cual únicamente dependen de las posiciones de las partículas. Dicho método es muy utilizado en simulaciones dinámicas ya que su esquema es realmente simple, y, puesto que se trata de un método simpléctico, posee una gran consistencia. Dicho esquema viene dado por

$$y_{n+1} = 2y_n - y_{n-1} + h^2 F(y_n) \tag{1.7}$$

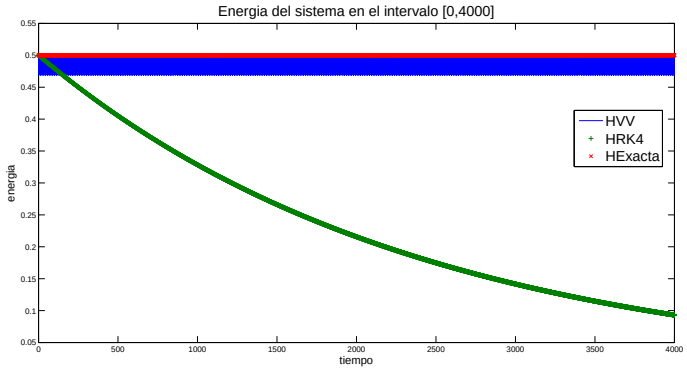


Figura 1.2: Gráfica donde se compara la energía del sistema respecto al tiempo para los métodos RK4 y Velocity Verlet (verde y azul, respectivamente), frente a la energía de la solución exacta (rojo).

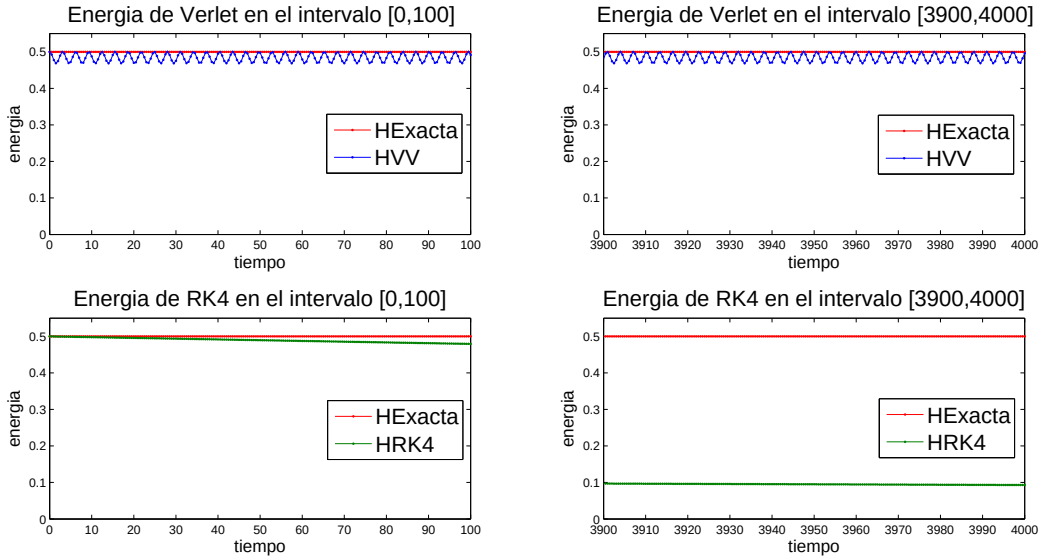


Figura 1.3: Comparación de gráficas más detallada de los resultados de la **Figura 1.2**.

el cual puede deducirse de (1.5) utilizando las diferencias finitas centradas en $O(h^2)$: Supongamos $y(t)$ solución del sistema, aplicando Taylor a las posiciones $y(t+h)$ e $y(t-h)$ se obtiene

$$\begin{aligned} y(t+h) &= y(t) + hy'(t) + \frac{h^2}{2}y''(t) + O(h^2) \\ y(t-h) &= y(t) - hy'(t) + \frac{h^2}{2}y''(t) + O(h^2) \end{aligned}$$

y, sumando dichos términos tenemos

$$y(t+h) + y(t-h) = 2y(t) + h^2 y''(t) + O(h^2)$$

de donde podemos aproximar la segunda derivada por

$$y''(t) \approx \frac{y(t+h) - 2y(t) + y(t-h)}{h^2}$$

y a partir de (1.5) deducir el método de Verlet (1.7)

$$y(t+h) \approx 2y(t) - y(t-h) + h^2 y''(t)$$

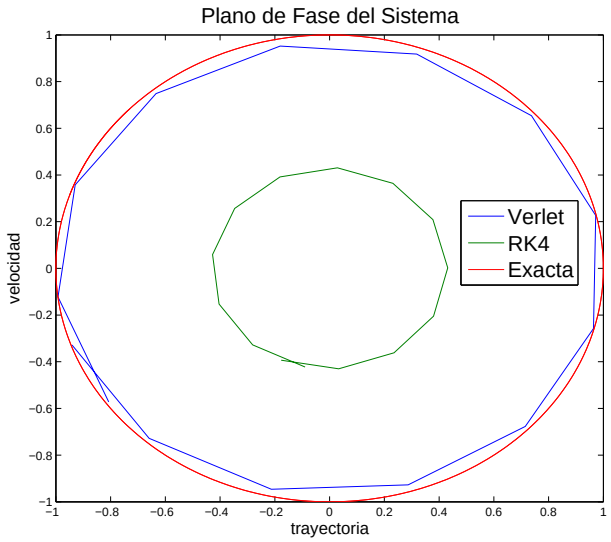


Figura 1.4: Plano de fase de la trayectoria y la velocidad del sistema para los métodos Verlet y RK4, junto con el de la exacta, para el último ciclo de cálculo.

sin necesidad de introducir la primera derivada en el sistema.

Así, puesto que hemos probado que el método Velocity Verlet es simpléctico y, por lo que acabamos de ver, existe una equivalencia entre éste y el método de Verlet, tenemos que este último también será simpléctico. Esto produce una gran ventaja a la hora de realizar simulaciones, pues tenemos un método capaz de conservar la energía del sistema que depende únicamente de la posición de las partículas. Esto será de gran relevancia, sobre todo, en el capítulo 2 donde veremos su utilidad en un caso práctico.

1.2.2. Métodos de Paso Adaptativo

Estos métodos consisten en una variación del paso h a partir del control de los errores de aproximación obtenidos a medida que se calculan las diferentes iteraciones del método. Así, si llegado el momento el error de aproximación supera la tolerancia establecida para éste, el método varía el paso para tener una mejor aproximación. Por el contrario, si el error es menor que la tolerancia, el método amplía el paso de tiempo con el fin de acelerar el rendimiento de los cálculos.

Veamos esta discusión más detalladamente basándonos, como referencia, en el libro [6, pág 293].

Sea $y(t)$ la solución al problema de Cauchy dado en (1.3) en un intervalo [0,T]. Fijado un paso h, supongamos que tenemos dos métodos de aproximación a dicha solución dados por

$$w_{n+1} = w_n + h\phi(t_n, w_n; h) \tag{1.8}$$
$$v_{n+1} = v_n + h\psi(t_n, v_n; h) \tag{1.9}$$

de forma que sean compatibles para poder determinar el error mediante los "pares encajados", y siendo de orden p y $\hat{p}$, respectivamente, tal que $\hat{p} \geq p+1$. Así, sus errores locales vienen dados

por:

$$\begin{aligned}\ell_{(1.8)}(t;h) &= y(t_{n+1}) - \underbrace{(y(t_n) + h\phi(t_n, y(t_n); h))}_{\hat{w}_{n+1}} \simeq \mathcal{O}(h^{p+1}) \simeq K_1 h^{p+1} \\ \ell_{(1.9)}(t;h) &= y(t_{n+1}) - \underbrace{(y(t_n) + h\psi(t_n, y(t_n); h))}_{\hat{v}_{n+1}} \simeq \mathcal{O}(h^{\hat{p}+1}) \simeq K_2 h^{\hat{p}+1}\end{aligned} \tag{1.10}$$

De modo que, para la solución exacta $y(t)$ tendríamos las aproximaciones de la **Figura 1.5**[3].

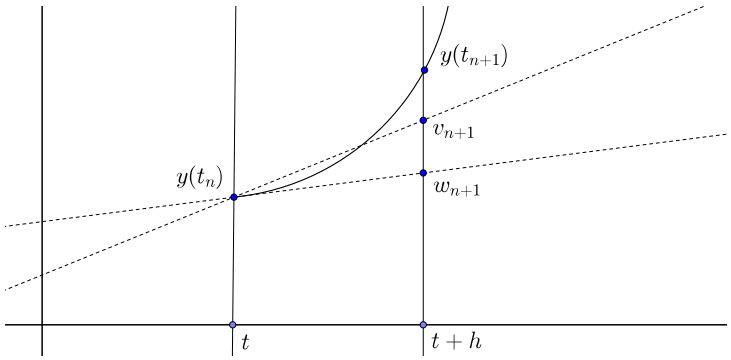


Figura 1.5: Aproximación de los métodos v y w a la solución exacta $y(t)$ en el paso t.

Veamos ahora cómo podemos controlar el error que se produce al realizar estas aproximaciones. Para ello comencemos considerando la diferencia entre ambas, la cual, podemos descomponer de la siguiente forma:

$$w_{n+1} - v_{n+1} = \underbrace{w_{n+1} - y(t_{n+1})}_{\text{error global con (1.8)}} + \underbrace{y(t_{n+1}) - v_{n+1}}_{\text{error global con (1.9)}}$$

Dividido de esta manera, observemos cómo acotar estos errores. Por un lado tenemos

$$w_{n+1} - y(t_{n+1}) = \underbrace{w_{n+1} - \hat{w}_{n+1}}_{\text{estimación de estabilidad}} + \underbrace{\hat{w}_{n+1} - y(t_{n+1})}_{\text{error de (1.10)}}$$

donde el error de (1.10) viene acotado por $K_1 h^{p+1}$, como podemos observar en la **Figura 1.6**, y la estimación de estabilidad viene acotada por una desigualdad tipo Lipschitz

$$\begin{aligned}|w_{n+1} - \hat{w}_{n+1}| &\overset{(1.8) \text{ y } (1.10)}{=} |w_n + \phi(t_n, w_n; h) - y(t_n) - h\phi(t_n, y(t_n); h)| \leq \\ &\leq |w_n - y(t_n)| + hL|w_n - y(t_n)| = (1 + hL)|w_n - y(t_n)|\end{aligned}$$

Sin embargo, caracterizando a estos métodos adaptativos, haremos una suposición más fuerte y consideraremos cierto que las aproximaciones w_n, v_n son similares a la solución en el punto t_n, es decir, $w_n \approx y(t_n) \approx v_n$. Por lo que

$$|w_{n+1} - \hat{w}_{n+1}| \leq (1 + hL)\cancelto{0}{|w_n - y(t_n)|} = 0 \tag{1.11}$$

Así, $|w_{n+1} - y(t_{n+1})| \leq K_1 h^{p+1}$ y, análogamente $|y(t_{n+1}) - v_{n+1}| \leq K_2 h^{\hat{p}+1}$. Ahora, como $\hat{p} > p$ entonces $K_1 h^{p+1} >> K_2 h^{\hat{p}+1}$ de forma que podemos concluir que

[3]Realizado con geogebra [2]

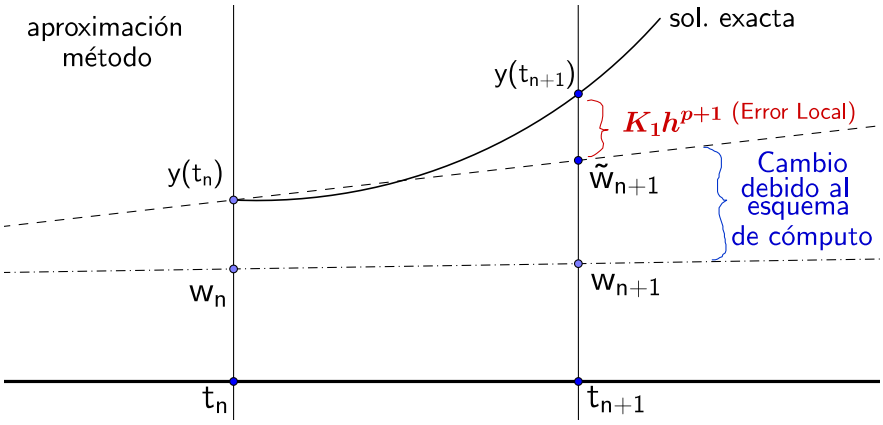


Figura 1.6: Gráfico de aproximación obtenido tomando como ejemplo el método (1.8).

$$\frac{1}{h}(w_{n+1} - v_{n+1}) \sim Kh^p =: \text{error local}$$

Visto esto tenemos ahora una cota, dependiente de h, para el error producido al comparar ambas aproximaciones de forma que podemos dar paso a estudiar el control de la adaptación del paso del método. Definamos

$$\tau(h) \approx \frac{1}{h}(w_{n+1} - v_{n+1}) \tag{1.12}$$

de forma que se estima mediante los cálculos realizados en el paso t para calcular el $t + h$ (**Figura 1.7**)

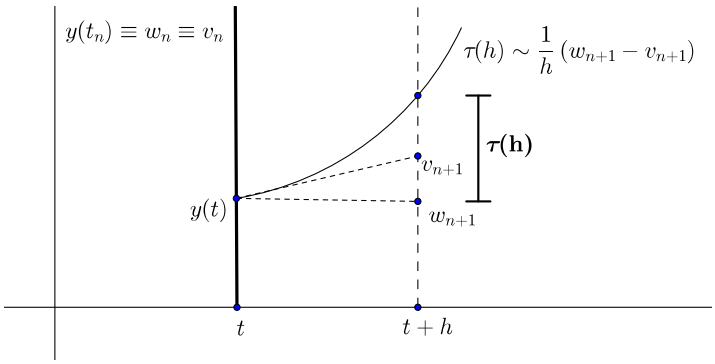


Figura 1.7: Esquema de adaptación de paso de los métodos v y w, en el tiempo t, con el error de estimación $\tau(h)$.

De esta forma se establece una tolerancia "*tol*", con el fin de no aceptar aproximaciones con un error en su aproximación mayor a dicha tolerancia, y un paso de tiempo máximo "$h_{\text{máx}}$" para los métodos, como cota superior de éstos a considerar. Así, si nos encontramos en el punto t y queremos avanzar hasta el $t+h$, en el caso de que $\tau(h_{\text{máx}}) < tol$ aceptamos como paso $h = h_{\text{máx}}$ y avanzamos hasta $t + h$ con el valor v_{n+1} (por ser el método de mayor orden). De lo contrario, si $\tau(h_{\text{máx}}) > tol$, entonces hemos de calcular un h' nuevo para avanzar. Tomando este h' como $h' = hq$, queremos que $\tau(h') < tol$, es decir, $\tau(qh) < tol$ pero $\tau(qh) \approx K(qh)^p = K(q^p h^p) = q^p(Kh^p) = q^p\tau(h)$. Así, si obligamos a que

$$\tau(h') < tol \implies q^p\tau(h) < tol \implies q < \left(\frac{tol}{\tau(h)}\right)^{\frac{1}{p}} \overset{(1.12)}{\implies} q < \left(\frac{h\, tol}{|w_{n+1} - v_{n+1}|}\right)^{\frac{1}{p}}$$

Tomando entonces $q = \left(\frac{1}{2}\frac{h\,tol}{|w_{n+1}-v_{n+1}|}\right)^{\frac{1}{p}}$ podemos aplicar el método con el nuevo h' y realizar nuevamente la comprobación.

Haciendo esta sutil comprobación lo que conseguimos es evitar esa dependencia inicial del paso h del método, dándonos una mayor libertad a la hora de inicializar los cálculos. No importa el paso inicial con que se comience, mientras que introduzcamos una tolerancia lo suficientemente pequeña, el propio método autoajustará el paso h para restablecerlo en los casos donde sea necesario, o usar uno más grande, para agilizar la computación mientras no lo sea.

Esquema del Método Adaptativo

Un esquema que represente este método adaptativo podemos obtenerlo de [6], el cual viene dado por el siguiente pseudocódigo:

Dado un Problema de Cauchy (1.1)

- Fijamos una tolerancia Tol, un paso máximo $h_{\text{máx}}$, uno mínimo $h_{\min}$, $n = 0$ y p, siendo éste el mínimo de los órdenes de los métodos del par encajado.
- Tomamos $h = h_{\text{máx}}$
- Determinamos $w_0 = v_0 = y_0$
- Mientras $t < T$
  - $w_{n+1} = w_n + h\phi_f(t_n, w_n, w_{n+1}; h)$
  - $v_{n+1} = v_n + h\phi_g(t_n, v_n, v_{n+1}; h)$
  - $R = \frac{1}{h}|w_{n+1} - v_{n+1}|$
  - Si $R < Tol$
    - $y_{n+1} = v_{n+1}$
    - $t = t + h$
    - $n = n + 1$
  - Si no,
    - $q = \left(\frac{1}{2} \cdot \frac{Tol}{R}\right)^{\frac{1}{p}}$
    - $h = qh$
  - Si $t + h \geq T$
    - $h = T - t$
  - Si no, si $h < h_{\min}$
    - Se excede el paso mínimo considerado, el método no puede aproximar la solución.

Características

Como hemos podido observar, el método adaptativo tiene la particularidad de emplear dos métodos, uno de orden mayor que el otro, con el fin de comparar y mejorar la aproximación. Esto puede generar una problemática a la hora de su computación ya que, para cada iteración, es necesario calcular las aproximaciones que produzcan ambos métodos. Esto implica que se debe realizar, para cada iteración, todos los procesos de un método y, tras éste, todos los del segundo, lo cual puede ralentizar la computación de cálculo final considerablemente.

Con el fin de resolver esta problemática, en lugar de utilizar dos métodos numéricos cualesquiera de distinto orden, se pretende encontrar una situación en la que el cálculo sea más eficiente. Por tanto, lo interesante será encontrar métodos en los cuales el cálculo de sus aproximaciones guarden cierta semejanza. Para ello, podemos recurrir a la noción de los pares encajados desarrollados por Fehlberg, reflejada en libros como Kincaid [8] o Quarteroni [16], donde es muy usual el empleo de métodos Runge-Kutta que se basen en dicha noción:

Fijado un método RK de orden p con el que vayamos a aplicar el método adaptativo, éste viene identificado por un tablero de Butcher de la forma

$$\begin{array}{c|c} c & A \\ \hline & \beta^T \end{array}$$

bastará considerar otro método RK de orden mayor tal que la identificación con su tablero de Butcher sea de la forma

$$\begin{array}{c|c} c & A \\ \hline & \hat{\beta}^T \end{array}$$

de forma que los coeficientes k_i, según la notación de (1.4), coincidan y, por cada iteración, sea suficiente con realizar la obtención de estos valores una única vez y calcular las aproximaciones con el vector b correspondiente.

Es el caso así de la familia de métodos adaptativos Runge-Kutta-Fehlberg[4]. En esta familia se utiliza, generalmente, el método RKF4(5) que hace uso de los métodos RK de orden 4 y 5, respectivamente, debido a que los tableros de Butcher de estos métodos coinciden, salvo en el vector β. Así, en lugar de realizar por separado los cálculos referentes a cada método podremos aprovechar los ya calculados para RK4 y emplearlos en RK5, agilizando así el proceso de cálculo.

Nuestro objetivo ahora es ver cómo tiene esto utilidad en problemas reales.

Shampine Flame

Acudamos ahora a un curioso ejemplo donde trataremos un problema rígido[5], correspondiente al encendido de una llama, problema físico y cotidiano en nuestros hogares cada día. En esta situación encontramos un cambio muy brusco en la ecuación que nos indica la temperatura respecto al tiempo trascurrido, ya que, llegados un punto y_*, pasamos de no tener prácticamente presencia de calor a obtener, de pronto, un alto nivel de temperatura.

En estos casos, y con un pico tan brusco, a los métodos numéricos les cuesta mucho hacer aproximaciones precisas de la solución. Observemos que antes y después del momento de prenderse la llama, la situación se mantiene relativamente constante (la aproximación es sencilla), pero en el punto de encendido encontramos un cambio de escala, en un intervalo tan pequeño, que la precisión debe ser lo suficientemente alta. Puesto que esta precisión viene determinada por el paso de tiempo h, los métodos no adaptativos pueden dar soluciones erroneas si no ajustamos mucho el paso, y, de lo contrario, si lo ajustamos demasiado, perdemos mucho tiempo

[4]Dichos métodos fueron desarrollado por Erwin Fehlberg, junto a otras técnicas de control de errores, mientras estaba al servicio de la NASA durante los 60, recibiendo en 1969 la medalla por los "Excepcionales Logros Científicos de la NASA" por su trabajo,[6, pág 296].

[5]Problema cuyas soluciones contienen nodos significativamente diferentes para la variable independiente.

computacional en los intervalos donde la función es, prácticamente, constante, lo cual no es eficiente.

Consideremos, por tanto, la ecuación diferencial que rige este comportamiento, dada por:

$$\begin{cases} y' = y^2 - y^3 \quad t \in [0, \frac{2}{y_0}] \\ y(0) = 0.0001 \end{cases} \tag{1.13}$$

Resolvamos ahora este problema utilizando una tolerancia de $Tol = 10^{-4}$, una delimitación del paso máximo y mínimo de h tal que $hmax = 0.25,\ hmin = 0.001$, y el siguiente tablero de Butcher.

c	A					
0	0	0	0	0	0	0
1/4	1/4	0	0	0	0	0
3/8	3/32	9/32	0	0	0	0
12/13	1932/2197	−7200/2197	−7296/2197	0	0	0
1	439/216	−8	−3680/513	−845/4104	0	0
1/12	−8/27	2	−3544/2565	1859/4101	−11/40	0
b orden 4	25/216	0	1408/2565	2197/4101	−1/5	0
b orden 5	16/135	0	6656/12825	28561/56430	−9/50	2/55

En este ejemplo utilizaremos, como método adaptativo, el método RKF4(5) para compararlo con otro método de paso fijo, el método Runge-Kutta(4). Para esta ecuación no existe solución analítica con la que podamos comparar el error, pero bastará observar el comportamiento relativo de cada método. Utilizando MATLAB y resolviendo el problema con los métodos Adaptativo y de RK4, obtenemos los resultados mostrados en la **Figura 1.8**

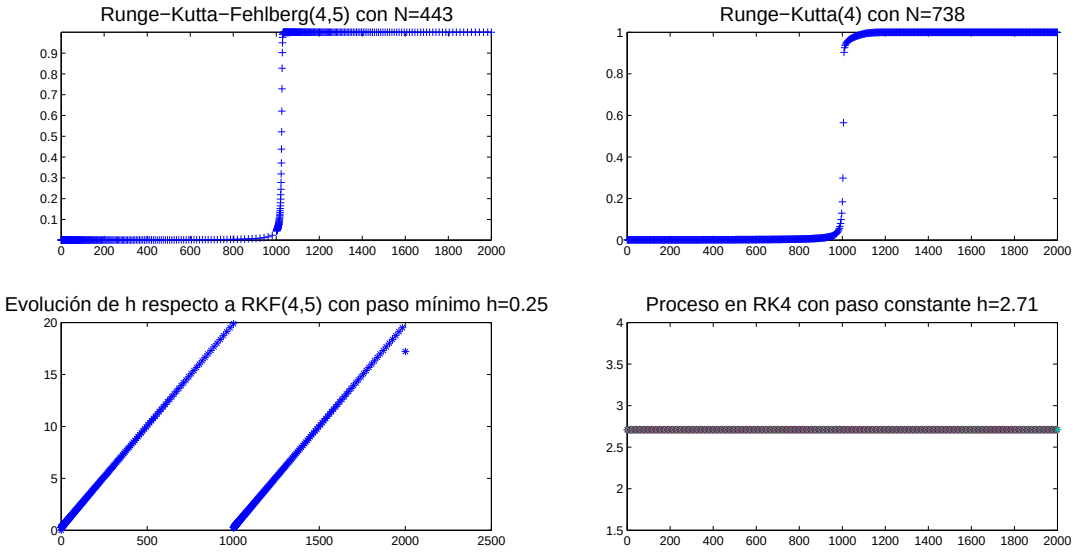


Figura 1.8: Ejemplo: Shampine Flame. Comparación RKF4(5) con RK4 para la condición inicial $y_0 = 0.001$

Como podemos observar en las gráficas, a pesar del alto orden del método RK4, al ser de paso fijo, necesitamos una partición muy fina para evitar que la aproximación explote al llegar

al punto crítico. Sin embargo, el método RKF4(5), adaptando el paso y haciéndolo mucho más grueso, permite en tan solo $N = 443$ iteraciones darnos una idea muy aproximada de la solución real y, además, utilizando como paso mínimo durante toda la iteración el máximo que se le había establecido.

1.2.3. Conclusión

El estudio realizado para estos dos nuevos tipos de métodos numéricos nos demuestra, en cierta medida, la gran utilidad que pueden tener, ya no solamente debido a la eficacia de resolución de algunos problemas frente a otros métodos de paso fijo, sino también ante la resolución de problemas rígidos (muy problemáticos en el estudio de las soluciones numéricas de ecuaciones diferenciales) o de problemas cuya solución se extienda durante un largo periodo de tiempo.

Con esto hemos podido ver que, además de los métodos usuales tratados en el Grado, el mundo de los métodos numéricos puede ser muy amplio y encontramos técnicas tan interesantes como las de los métodos adaptativos, los cuales, con un pequeño estudio del error local producido en cada iteración nos permite hacer aproximaciones muchísimo más precisas en situaciones donde otros métodos explotan; o los simplécticos que, gracias a la conservación del hamiltoniano constante, o con valores muy cercanos a dicha constante, permiten mantener la estabilidad del método. Esto hace que sean realmente demandados a la hora de realizar simulaciones gráficas ya que, éstas, pretenden llevarse a cabo durante periodos de tiempo prolongados.

En este capítulo nos hemos centrado en el estudio de los métodos adaptativos y los simplécticos, junto con la mejora que producen con algunos sistemas. Sin embargo, en todo este ámbito, queremos destacar antes de acabar el capítulo y, remarcando la ya citada amplitud de este tema, que también podemos encontrar muchos otros métodos y procedimientos. Entre otros, se encuentran los ya citados multipaso o, incluso, los métodos implícitos, que puede desarrollar grandes ventajas según qué tipo de problema estemos tratando al considerar, en el cálculo de una nueva posición, información sobre ésta misma.

Con todo ello, damos paso a una de las aplicaciones prácticas más curiosas, de las cuales ya hemos comentado algo en el capítulo: procedimientos numéricos en la simulación de procesos físicos. En el siguiente apartado trataremos un problema real de la computación, donde comprobaremos el uso de la nueva versión del método de Verlet citada, cuyo propósito refuerza la conclusión que ya habíamos explicado sobre el objetivo de los métodos simplécticos: la eficiencia de la aproximación de soluciones ante tiempos de computación extensos gracias a la conservación del Hamiltoniano.

Capítulo 2

Simulación en 3D

"La pantalla es una ventana
a través de la cual uno ve un mundo virtual.
El desafío es hacer que ese mundo se vea real,
actúe real, suene real, se sienta real."
— Ivan Sutherland.

2.1. Enfrentamiento a un problema computacional

Una de las grandes cualidades que aprendemos en este Grado de Matemáticas es, como matemáticos, la capacidad de enfrentarnos a problemas propuestos y la destreza para abstraernos, estructurarlos, y poder, al fin, resolverlos. Tanto es así que una de nuestras competencias en este grado es: *"CGM-4.Saber abstraer las propiedades estructurales (de objetos matemáticos, de la realidad observada, y de otros ámbitos) distinguiéndolas de aquellas puramente ocasionales y poder comprobarlas con demostraciones o refutarlas con contraejemplos, así como identificar errores en razonamientos incorrectos"*. En este caso nos vamos a encontrar con una situación de esta índole.

En este capítulo se nos propone enfrentarnos a un código en C. Este código genera una simulación en 3D en la que podemos observar la interacción de una esfera y un paño, este último suspendido de dos de sus extremos. La dinamización de esta simulación se basa en una técnica de computación numérica. Nuestra labor en este capítulo será la de ser capaces de entender el programa, saber cómo funciona, ser capaz de modificarlo, mejorándolo si es posible, e implementar nuevos métodos numéricos a fin de poder contrastar su eficacia.

Todo ello nos ayudará a comprender cómo afectan los métodos numéricos, explicados en el capítulo 1, al proceso de una simulación real, y observar cómo, en ocasiones, los cambios pueden ser realmente significativos entre unos y otros métodos. Podemos remarcar así, en este contexto, otras de las competencias del Grado como son: *"CGM-7. Proponer, analizar, validar e interpretar modelos de situaciones reales sencillas, utilizando las herramientas matemáticas más adecuadas a los fines que se persigan."*, *"CGM-8. Utilizar aplicaciones informáticas de análisis estadístico, cálculo numérico y simbólico, visualización gráfica, optimización u otras para experimentar en Matemáticas y resolver problemas."* o *"CGM-9. Desarrollar programas que resuelvan problemas matemáticos utilizando para cada caso el entorno computacional adecuado."*

2.2. Presentación del Programa

Comencemos por presentar el problema al que nos enfrentamos. Como hemos dicho se trata de una simulación en la que una esfera interactúa con un paño suspendido en el aire, con dos de sus extremos fijos. Este paño se ve afectado por varias fuerzas: la de gravedad, la fuerza del viento y el movimiento producido al chocar con la pelota. De esta forma el paño no está estático, sino que está en continuo movimiento produciendo ondas o desplazándose a lo largo de la pantalla. La posición de la pelota también puede verse alterada por el usuario, de manera que la confrontación de una con el otro puede realizarse desde una gran variedad de ángulos distintos. Además, podemos cambiar tanto la dirección como la fuerza del viento y, también, la vista de visualización de la escena, con el fin de observar la dinámica de la situación desde diferentes puntos. Tanto el movimiento de la esfera como los cambios de dirección del viento o la cámara, se realizan mediante la selección de la opción deseada en el menú, que se genera al pulsar el botón derecho del ratón sobre la escena. Esto da como resultado lo que podemos observar en la **Figura 2.1**

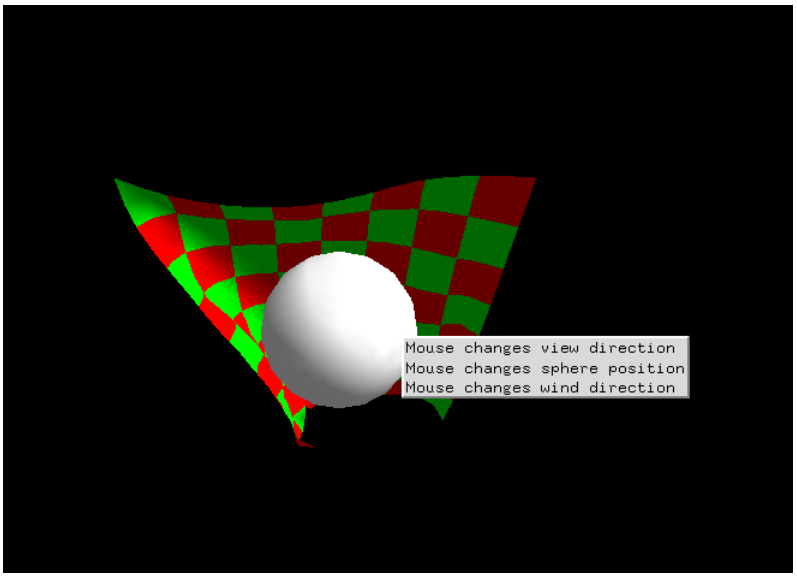


Figura 2.1: Pantalla principla de la simulación original.

Este problema tiene claramente una base matemática. Se trata de un problema físico basado en la ecuación del movimiento de un sistema de partículas bajo la intervención de ciertas fuerzas externas. El desplazamiento de dichas partículas se genera gracias a la Segunda Ley de Newton, [17, pág 115]. Si denotamos por $\vec{y}_p(t)$ la posición de una partícula p en un tiempo t y por $\vec{a}_p(t)$ su aceleración en dicho tiempo, entonces, su movimiento viene determinado por la solución de la ecuación diferencial que satisfaga:

$$\vec{y}_p''(t) = \vec{a}_p(t) = \frac{F(t, \vec{y}(t), \vec{y}'(t))}{m} \tag{2.1}$$

Donde $F(t, \overrightarrow{y}(t), \overrightarrow{y}'(t))$ denota la fuerza resultante obtenida como suma de las fuerzas que actúen sobre la escena y, m, la masa de la partícula. Dicha solución vendrá unívocamente precisada por las condiciones iniciales donde sea inicializada la partícula. El objetivo que tiene el análisis de este proceso físico es el de poder aproximar la trayectoria exacta de la partícula a partir de técnicas computacionales, basadas en diferentes métodos numéricos.

En este caso nos encontramos ante una ecuación diferencial de segundo orden, de modo que, para enfrentarnos a ella, optaremos por la opción más generalizada y la trasformaremos en un

sistema de primer orden para poder trabajar con ella cómodamente. Para esto, bastará añadir la derivada $\vec{y}'_p(t) = \vec{v}_p(t)$ a la ecuación (2.1) quedando:

$$\begin{cases} \vec{y}'_p(t) = f(t, y(t), v(t)) = \vec{v}_p(t) \\ \vec{v}'_p(t) = g(t, y(t), v(t)) = \frac{F(t, \vec{y}(t), \vec{y}'(t))}{m} \end{cases} \tag{2.2}$$

A partir de aquí sí que podemos trabajar con este sistema y aplicar los métodos numéricos, de la forma adecuada, para calcular las trayectorias aproximadas de las partículas.

Asignando uno de estos sistemas de ecuaciones a cada una de las partículas p_i que conforman el paño, podemos obtener un sistema global que las relacione todas. De esta manera poder simular, en conjunto, todas ellas con el fin de alcanzar la sensación de un paño completo tal y como podemos observar en la **Figura 2.1**.

2.2.1. Secuencia del Programa

Con el fin de tener una visión global del funcionamiento del código del programa, describimos, a continuación, un pequeño diagrama que lo resuma. En él, indicaremos aquellas funciones prescindibles para obtener resultados básicos, así como aquellas que son necesarias para el propio funcionamiento de la librería.

- Función `main`
  - Inicializa la ventana, entorno de `GLUT` y el modo `display`.
  - `Init()`
    - `Initializecloth()`
      - Inicia { - Posición aproximada; - Velocidades; - Posición exacta
      - Inicia arrays auxiliares { - `cloth2`; - `vel2`
      - Establece los punteros
    - Habilita las funciones para la textura (Se puede eliminar si no se quiere definir ninguna textura)
    - Habilita la iluminación { (Si se elimina desabilita todas las iluminaciones); (Se puede eliminar siempre y cuando se asignen colores a los elementos que queramos dibujar)
  - `Idle()`
    - `Updatecloth()`
      - `accumulateforces()` ; (Calcula las fuerzas para cada instante)
      - `verlet()` ; (Aplica el método numérico)
      - `satisfyconstraints()` ; (Corrige las posiciones)
      - `calculatenormals()` ; (Calcula las normales)
    - `glutPosRedisplay()` ; (Continúa el `MainLoop` y prepara la escena para el nuevo muestreo)

- `display()`
  - Coloca la vista de visualización.
  - Renderiza la escena.
  - `drawcloth()`
    - ◇ Dibuja los puntos aproximados
  - `glutSwapBuffers()` (Permite la visualización de la escena actualizando el buffer)
- `Reshape()` (Remodela y especifica los elementos de la escena para poder verlos por pantalla)
- `mousemove()` y `mousedown()` (Se pueden eliminar, siempre y cuando no queramos interactuar con el ratón en la pantalla)
- `keyboard()` (Se puede eliminar si no se desea tener interacción con el usuario mediante el teclado)
- `MainLoop()` (Sin ella no es posible establecer el entorno de trabajo de `OpengL` de forma que permita las interacciones entre las funciones de dinamización del programa. Sin ella no es posible, si quiera, su ejecución

2.3. Comprensión del Código

Una vez presentado el programa nos disponemos a escudriñar el código para comprenderlo y sacarle todo el jugo posible.

La primera gran dificultad a la que nos enfrentamos es, en principio, la programación. En nuestro Grado tenemos dos asignaturas enfocadas directamente a este tema: *Introducción al Software Científico y a la Programación*, y *Programación Orientada a Objetos*. Sin embargo, ambas asignaturas están enfocadas a la programación en Java, y C, aunque similar, tiene grandes diferencias. Aún así, haber cursado estas dos asignaturas nos ha ayudado en gran medida pues, la utilización de las variables, sus diferentes tipos, la composición y distribución de las funciones a lo largo de un programa, llamadas y referencias... Todo ello, a grandes rasgos, se organiza de la misma forma en ambos. Sin embargo, a continuación resaltaremos algunas diferencias relevantes:

La primera, y más significativa, diferencia que encontramos entre estos dos lenguajes es su funcionalidad. Para llevar a cabo esta distinción tomaremos como referencia principal el libro guía [14]. C es un lenguaje de programación imperativa, es decir, describe la programación en términos del estado del programa y sentencias que cambien dicho estado. En contraposición, Java es un lenguaje de POO, el cual, hace uso de objetos que manipulan los datos de entrada para la obtención de datos de salida específicos[1].

Por otro lado, otra de las grandes diferencias que podemos encontrar, y que otorga a C la velocidad de computación que tiene, es el uso de punteros. Un puntero es una variable que contiene una dirección de memoria de otra variable a la que "apunta", por ello, el acceso a dicha memoria es mucho más rápido, lo que permite agilizar enormemente la computación. Los punteros se caracterizan, en su declaración, por ir precedidos de un asterisco "∗", de esta manera C reconoce que esta variable, además de ser del tipo de primitiva que sea, es un puntero. Dicho puntero se debe vincular más tarde a la variable que queramos que apunte, haciendo, simplemente, una asignación de la variable apuntada. Java, sin embargo, hace uso de referencias que incluyen a todo un objeto, el cual, tendrá asignada su dirección de memoria correspondiente.

[1] Ambas definiciones las podemos en contrar en https://es.wikipedia.org/wiki/Programacion_imperativa y https://es.wikipedia.org/wiki/Programacion_orientada_a_objetos, respectivamente.

Otra gran diferencia es la referida a la distribución de las funciones en el código. En Java, la declaración y construcción de éstas a lo largo del programa no influye en la compilación del mismo pero, en C, esto no es así. Para que la función principal "`main`", o cualquier función construida a lo largo del código, reconozca otra función declarada en éste, dicha función habrá de ser creada antes de su primera llamada o, de lo contrario, la compilación producirá un error por no reconocerla. Para evitar esto entran en juego las llamadas "funciones prototipo", que son declaraciones que se realizan de la función al principio del código indicando que, posteriormente, serán construidas. De esta forma, al compilar, C reconoce que estas funciones van a ser declaradas y no hemos de preocuparnos de hacerlo antes de su primera llamada, lo cual, en códigos muy extensos, puede ser un problema.

Además de las características propiamente referidas a la programación en sí, C también varía de Java en otros aspectos. Las primeras líneas de todo código en C van referidas siempre a las llamadas efectuadas a las librerías que el programa a desarrollar va a necesitar. Estas líneas vienen caracterizadas por el comando "`# include`"

Como hemos podido comprobar a lo largo de los cursos cada vez que hemos hecho uso de Java, éste contiene numerosas librerías estándar ya asumidas implícitamente, de forma que no es necesario incluirlas en el código y, si necesitamos hacer uso de ellas, bastará con utilizar el comando correspondiente. Sin embargo, en C, para muchos usos básicos es necesario incluir la librería correspondiente de manera explícita para que nos permita hacer lo que precisemos, ya que, C, no lleva asumido apenas librerías en su compilación estándar. Esto le permite ser un programa más básico y dar una manejabilidad diferente a la de Java que, en muchas ocasiones, es preferible para los programadores a la hora de realizar programas complejos, de visualizaciones dinámicas (como es nuestro caso), o relacionados con la base de datos de algún software.

Para comprobar todo esto, en http://members.shaw.ca/jordanisaak/graphics.htm podemos encontrar el código fuente en el que nos basamos[2]. Puesto que, en nuestro caso, se trata de un programa con fines gráficos en 3D, el código entero se apoya en una interfaz llamada *OpenGL* y su librería GLUT. Parafraseando a la página principal de esta interfaz[3] y al libro [9]:

"OpenGL es el entorno principal para el desarrollo de aplicaciones gráficas interactivas en 2D y 3D. Este entorno alberga innovaciones y desarrollos de aplicaciones ágiles incorporando un amplio conjunto de funciones de renderización, mapeo de texturas, efectos especiales y otras potentes funciones de visualización. OpenGL es ya ampliamente aceptado en la industria estándar y es usado por muchas (si no todas) las casas de producción profesional. No es un lenguaje de programación sino una API, por ello, es capaz de aportar todo lo necesario para la comunicación entre nuestro software y los gráficos de nuestro sistema, gracias a la potente librería GLUT. Ésta elimina la necesidad de comprender la programación básica de Windows, por lo que nos permite enfocarnos, únicamente, en las salidas gráficas."

Como podemos observar, queda evidenciado la gran capacidad y funcionalidad que tiene este entorno en nuestro código, pues la gran cantidad de funciones propias de esta librería es abrumadora. Así, nuestro primer paso será analizar cada una de las funciones por las que está compuesto el programa para saber qué labor tienen y, a partir de aquí, saber cómo está constituido y desarrollado.

2.3.1. Funciones Generales y Específicas

El código se divide en dos grandes tipos de funciones:

[2] Código original llevado a cabo por Jordan Isaak, resto de información en la página web citada al comienzo del código

[3] https://www.opengl.org/

- **Funciones Generales.-** Estas funciones están definidas y desarrolladas por el programador. Se encargan de fragmentar y ordenar el código de manera que permitan su dinamización a través del compilador. Algunas ayudan a que el programa no esté constituido por una única función principal con numerosas líneas repetitivas. Otras, en cambio, es necesario declararlas pues, la propia librería GLUT, necesita de éstas para su correcto funcionamiento.
- **Funciones Específicas.-** Llama la atención que, dentro de las funciones generales, hay muchas funciones propias del entorno OpenGL que tienen un cometido concreto. Estas funciones no las determina el programador, sino que tienen su propia estructura interna definida en la librería GLUT. Algunas podemos deducirlas por el contexto pero, mediante las referencias [14], [3] y [1], en el **Anexo 2** podemos encontrar una lista detallada en la que queda resumido el papel que desempeña cada una de estas funciones y, en los casos donde sea necesario, la explicación de la variación que produce el empleo de uno, u otro, de los parámetros propios de ésta.

Veamos ahora, más detenidamente, cuáles son estas "Funciones Generales" en nuestro código particular.

Antes de sumergirnos directamente en la programación, conviene hacer mención y destacar algunos elementos propios de este código.

- *Elemento Primitivo.-* Cuando nos referimos a un elemento primitivo aludimos a los elementos declarados mediante los tipos de datos "`char`", "`int`", "`bool`", "`float`", "`double`",...
- "`struc`".- Es una estructura semejante a la "Clase" en Java. Permite definir elementos primitivos y propiedades concretas de estos con el fin de poder crear, a lo largo del programa, otros elementos más complejos basados en la estructura definida.
- *Escena.-* Llamamos escena al resultado de la renderización por pantalla que ejecuta el programa.
- *Matriz Modelmatrix.-* La matriz Modelmatrix es una matriz que guarda las posiciones del conjunto de vértices de la escena y nos permite desplazarlos a través de ésta. Cuando nosotros creamos un objeto, dicho objeto se encuentra determinado en unas posiciones de "Espacio Modelo". La matriz Modelmatrix es la herramienta que nos permite pasar de este "Espacio Modelo" al "Espacio Real" y que la escena, para nosotros, tenga sentido.
- *Matriz Modelview.-* Al igual que la matriz anterior controla la posición de los elementos de la escena, la matriz Modelview controla la posición de la cámara respecto a las primitivas que renderizamos.
- *Matriz Projection.-* Esta matriz va muy vinculada a la anterior pues se encarga de determinar la perspectiva aplicada a las primitivas al pasar del "Espacio Modelo" al "Espacio Real" para una correcta visualización de la escena, es decir, proyecta las transformaciones de la matriz modelmatrix de la escena de 3 a 2 dimensiones.
- *ASCII (American Standard Code for Information Interchange).-* El código ASCII es un código estándar de caracteres basado en el alfabeto latino compuesto por números, letras mayúsculas y minúsuclas, y otros símbolos comúnmente usados.

Una vez conocidos estos elementos veamos la distribución de las funciones del programa. Con el fin de tener una perspectiva general, pero más exhaustivo que en el esquema del punto 2.2.1. del funcionamiento del código comencemos describiendo la utilidad de las Funciones Generales.

Funciones Generales

Siguiendo, en orden, las líneas de código base que nos encontramos en la página del programador (citada en la página 23) tenemos, en primer lugar, la inclusión de las librerías que necesitamos: `#include <glut.h>`, `#include <math.h>` e `#include <stdio.h>`; y la declaración de la clase `CONSTRAINT` y de las constantes y variables que necesitaremos a lo largo del programa. Para el almacenaje de los datos por los que se compone la tela se hace uso, fundamentalmente, de dos arrays: `cloth1` y `cloth2`. Éstos nos servirán para guardar las coordenadas de las partículas del instante actual y el inmediatamente anterior, respectivamente. Dichos arrays tienen una dimensión de cuatro veces el número de partículas que configuran la tela. Esto es debido a que cada partícula necesita cuatro referencias de memoria, las destinadas a las coordenadas (x, y, z) y a la inversa de la masa, siendo esta última 1 para todas las partículas, salvo para las fijas que se les asigna 0 (masa infinita) a fin de que, en los cálculos posteriores se multiplique por ella y su resultado sea nulo. Así, en un sistema, las coordenadas de las dos primeras partículas se distribuirán de la siguiente manera:

x_1	y_1	z_1	$\frac{1}{m_1}$	x_2	y_2	z_2	$\frac{1}{m_2}$
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

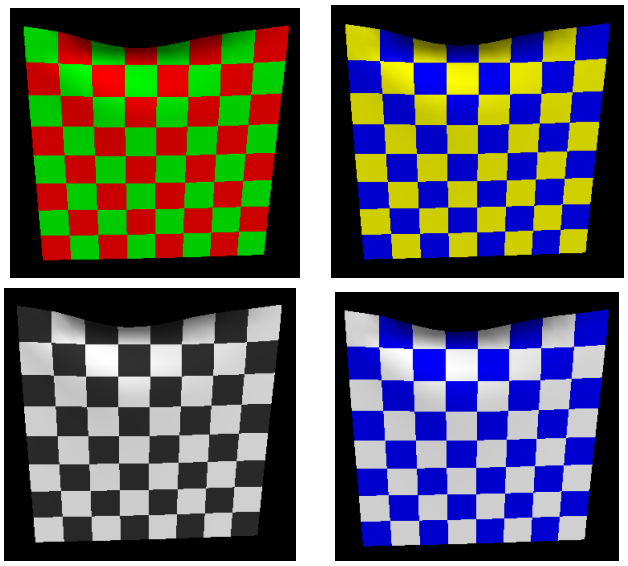
Posteriormente, y a fin de agilizar la renderización de la simulación, dichos arrays se vincularán a los punteros `currentcloth` y `previouscloth`, respectivamente.

Tras esto, nos encontramos una primera función llamada `generatetexture()` a la cual no se le pasan argumentos ni devuelve valores. Esta función, como su nombre indica, se encarga de generar la textura sobre la superficie del paño con el que vamos a trabajar. Inspirándonos en [9] podemos decir que las texturas son imágenes que, aplicadas a objetos del entorno de OpenGL mediante un mapeo de píxeles, cambian las características de la superficie de éste y hacen que el modelo sea mucho más realista sin aumentar el número de polígonos que lo conforman. Se encargan, por tanto, de dar la sensación de "realidad" al objeto de la simulación que se realice. Además, como se especifica en el **Anexo 2**, se puede asignar una iluminación asociada a las caras que conforman la textura, distinguiendo entre frontal y posterior. De esta forma, estableciendo una fuente de luz en la escena, y una iluminación a la textura, la sensación de realismo en la pantalla aumenta.

Su estructura se basa en tres condicionales, de forma que, según el valor que tome la variable `currenttexture` (0, 1, 2 ó 3), la texturización de la superficie se elaborará de una manera diferente. El proceso de asignación de textura se realiza atribuyendo valores de RGB al array `texture`. Como ejemplo, la asignación de valores para `currenttexture` = 0 se lleva a cabo tal y como vemos a continuación:

```
if(currenttexture == 0) {
        for(int i = 0; i < TEXTURE_SIZE; i++) {
                for(int j = 0; j < TEXTURE_SIZE; j++) {
                        texture[3 * (i * TEXTURE_SIZE + j)] = ((i + j) % 2) *
                            255;
                        texture[3 * (i * TEXTURE_SIZE + j) + 1] = 255 - ((i + j)
                            % 2) * 255;
                        texture[3 * (i * TEXTURE_SIZE + j) + 2] = 0;
                }
        }
}
```

Tomando modulo 2 de la suma de la coordenadas en las que se divide, visualmente, la tela, podemos establecer una cuadrícula de 2 colores que genera el resultado de la **Figura 2.1**. Así, podemos encontrarnos con los siguientes 4 tipos:



Tras esto, se llama a algunas de las librerías específicas de `OpenGL`, explicadas en el **Anexo 2**. Éstas se encargarán de elaborar y fijar la textura en la superficie determinada por el parámetro `texture` y hacerla visible.

Seguidamente, aparece la función `accumulateforces`. Al igual que la anterior, esta función carece de argumentos y tampoco devuelve ningún valor. Su labor es encargarse de calcular la aceleración de cada una de las partículas del sistema a partir de los datos proporcionados por la constante de la gravedad (`gravityforce`), la fuerza proporcionada por el viento, en caso de haberla, y la fuerza de rozamiento. Estas dos últimas a través de la información que aportan las normales de los puntos, sus velocidades y su masa.

A continuación, encontramos la función principal que se encarga del movimiento de las partículas, la función `verlet`. Siguiendo la dinámica de las anteriores, esta función no recibe ni devuelve valores. Su estructura se basa en el método de Verlet, ya citado en la página 9. El interés de haber utilizado este método se debe a que, como ya indicamos, es simpléctico y, además, depende únicamente de la posición de las partículas. Sin embargo, si necesitamos conocer su valor será suficiente con recurrir a una descomposición de Taylor para su aproximación. Únicamente basándonse en las posiciones del instante inmediatamente anterior y del actual, en el que nos encontremos, es capaz de aproximar la solución exacta. Recordando el esquema

$$\vec{y}_{p,n+1} = 2\vec{y}_{p,n} - \vec{y}_{p,n-1} + h^2 f(t_n, \vec{y}_n)$$

Se describe la función

```
void verlet(void) {
        float *tempfloatptr;

        //Update each particle via verlet integration
        for(int i = 0; i < numparticles; i++) {
                previouscloth[4 * i] = 2.0 * currentcloth[4 * i]
                                                - previouscloth[4 * i]
                                                + forces[3 * i] * currentcloth[4
                                                    * i + 3] * TIME_FACTOR;
                previouscloth[4 * i + 1] = 2.0 * currentcloth[4 * i + 1]
                                                - previouscloth[4 * i + 1]
                                                + forces[3 * i + 1] *
                                                    currentcloth[4 * i + 3] *
                                                    TIME_FACTOR;
                previouscloth[4 * i + 2] = 2.0 * currentcloth[4 * i + 2]
                                                - previouscloth[4 * i + 2]
```

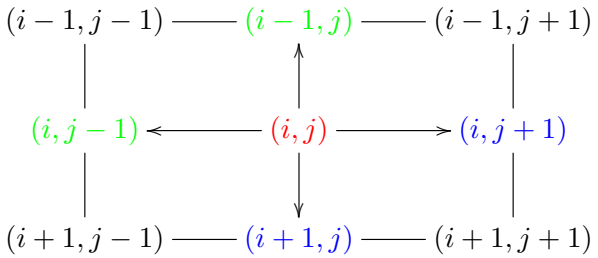
```
                                                    + forces[3 * i + 2] *
                                                        currentcloth[4 * i + 3] *
                                                        TIME_FACTOR;
        }

        //Update the current and previous cloth pointers
        tempfloatptr = previouscloth;
        previouscloth = currentcloth;
        currentcloth = tempfloatptr;
}
```

que reutiliza el puntero `previouscloth` como puntero auxiliar, puesto que la información actualizada tras cada iteración no se utiliza en la siguiente. Esta "reutilización" evita crear otro array al que tener que pasar la información de todas las coordenadas, pues recordemos que la cuarta dirección de memoria correspondiente a cada partícula va referida a la inversa de la masa de ésta y, copiar esta información en cada iteración no es eficaz puesto que es repetitiva. Finalmente, mediante la agilidad de la asociación que permite el puntero `tempfloatptr`, la actualización de los arrays es muchísimo más rápida que establecer directamente el array solución en el bucle.

La siguiente de las funciones que nos aparece en el código, `satisfyconstraints`, es la referida a las restricciones, definidas al comienzo el programa. Esta función ni tiene argumentos ni devuelve ningún valor, sino que su papel es el de suplir la propiedad que tendrían las partículas si, entre ellas, existiese una fuerza de atracción que las mantuviese unidas. Este programa no desarrolla una función específica que estructure y recorra las partículas para efectuar entre ellas la, usualmente utilizada en estas simulaciones, fuerza de atracción de un muelle. A grandes rasgos, lo que pretende esta función es establecer una distancia máxima entre las partículas proporcional a su distancia inicial. De esta manera, dar la sensación visual de que existe una fuerza que las mantiene unidas. Del mismo modo, esta función establece unas restricciones para la interacción de las partículas del paño y de la esfera, de forma que la distancia de éstas al centro de la esfera no pueda ser menor que su radio, de ser así, la posición de la partícula que incumpla esta restricción se verá alterada para quedar fuera del radio.

Tras las restricciones se declara la función `calculatenormals` sin argumentos ni valores devueltos. Esta función se encarga de calcular la normal de cada uno de los vértices, como suma resultante de las normales calculadas a partir de la siguiente estructura. Consideremos una partícula situada en la posición (i,j), entonces, su situación será semejante a la siguiente:



Ante esta situación podemos obtener normales a partir del producto vectorial de los vectores generados del nodo central (rojo) a los verdes, y del central a los azules. Esto se realiza mediante un bucle siguiendo el siguiente esquema:

Al recorrer todos los nodos con este algoritmo (primero de la forma de la izquierda y, después, con la de la derecha), como normal asociada al nodo (i, j), tendremos la suma de todas aquellas normales en las que entra en juego, tal y como presentábamos en el esquema anterior. La suma de estas normales dará lugar a la normal resultante que será la utilizada en los cálculos del resto de funciones que la requieran. En los casos de los nodos situados en los extremos, sólo podrá realizarse el cálculo de una de ellas, por la imposibilidad del cálculo de ambas, de manera que, directamete, ésa será la normal resultante para dichos nodos. Finalmente, la función normaliza los vectores normales obtenidos a fin de que en los cálculos posteriores la normal únicamente indique la dirección en la que se realiza la operación pertinente, y no afecte, de manera cuantitativa, al cálculo.

A continuación, nos encontramos con la función `updatecloth` que se encarga, principalmente, de llamar a las funciones ya descritas `accumulateforces()`, `satisfyconstraints()` y `calculatenormals()`.

Y, tras ésta, la función `initializecloth`. La estructura de esta función es algo más compleja que las anteriores, al llamar a esta función sí que es necesario pasar un argumento en su llamada, el entero `setuptype`. Esta variable tomará los valores 0 ó 1, y señalará la posición en la que se iniciará el paño: si `setuptype` =0, entonces el paño se inicializa en posición horizontal justo encima de la esfera; si, por el contrario, `setuptype` =1, entonces el paño se inicializa de forma vertical tras ésta. Una vez establecida la posición inicial del paño, se copian estas variables en un segundo array de posiciones, que nos servirá, más adelante, para guardar las posiciones de las partículas en el instante inmediatamente anterior al que nos encontremos y, así, poder utilizar esta información en funciones como la de `verlet`. Asimismo, asociamos los punteros de las posiciones a los dos arrays de posiciones que acabamos de inicializar. Por último, se establece una distancia de restricción entre las partículas, `restlength` (distancia lateral) y `diagonallength` (distancia diagonal), y se inicializan las restricciones que se utilizarán en la función `satisfyconstraints`.

Posteriormente, nos encontramos con la función `drawcloth`. Ésta, sin requerir argumentos ni devolver información, se encarga, a través de las funciones específicas de `GLUT` responsables de dibujar (descritas en el **Anexo 2**), de representar el paño con las posiciones de las partículas como información. En este caso, el parámetro para `glGlut` que estamos utilizando es `GL_TRIANGLE_STRIP`, de forma que, en la malla generada por las partículas, se generará una banda de triángulos que darán la sensación de la tela que observamos en la simulación.

Seguidamente, encontramos la función `menu` que asigna, mediante el argumento que se le pasa, el valor de la selección marcada con el ratón en el menú de la pantalla que éste genera, así como la función `idle`. Esta última tampoco devuelve ningún valor ni requiere de ningún argumento. Se encarga de sincronizar los tiempos de la máquina y de la visualización de la simulación. En este caso, la selección de menús con el ratón o de cambios de renderización con el teclado hacen que las llamadas entre las funciones del programa se alteren. Esto puede producir que se produzca un desfase en cuanto a la función que calcula las posiciones de las partículas con la función que las dibuja por pantalla. Para ello, en esta función se comparan el tiempo `currenttime`, que recibe los milisegundos de desfase desde que se llamó a la función `glutInit`, con el tiempo `simulationtime` que es el referente al tiempo t_n que utiliza nuestro método numérico. De esta forma, el programa entra en bucle de actualización de posiciones hasta que el valor de `simulationtime` alcance al de `currenttime`. Declarando todo esto en una función se obtiene

```
void idle(void) {
        currenttime = glutGet(GLUT_ELAPSED_TIME);
```

```
        //Update the simulation to match the current time
        while(simulationtime < currenttime) {
                updatecloth();
                simulationtime += TIME_STEP * 1000;
        }
        glutPostRedisplay();
}
```

La siguiente función va directamente relacionada con la función `initializecloth`, se trata de la función `init`. En ella, la primera acción es llamarla, pasándole como argumento `setuptype` = 0. Tras esto, continúa ejecutando algunas funciones para establecer la configuración de la textura del paño y la iluminación de la escena.

A continuación nos encontramos la función `display` que será una de las funciones que, mediante el comando `MainLoop()`, se irá rellamando continuamente a fin de poder ir visualizando los cambios gráficos producidos en la escena de la simulación. No necesita argumentos ni devuelve valores, simplemente actualiza la visión de la escena, renderiza el buffer y se encarga de dibujar los elementos en la pantalla. Para ello, en primer lugar realiza una llamada a la función `drawcloth()`, explicada anteriormente, tras lo cual dibuja la dirección del viento con una longitud proporcional a la magnitud que tenga, si existe, y, finalmente, dibuja la esfera. Al final de la función se actualiza el buffer con todos los cambios realizados.

Tras ésta se declara la función `reshape`, contenida en el bucle anterior mencionado (`MainLoop`). Esta función no devuelve nada pero sí es necesario pasarle dos enteros como parámetro referidos al ancho y alto de la pantalla, respectivamente. La estructura de esta función se basa, completamente, en funciones específicas de la librería `GLUT`, a través de las cuales remodela y actualiza los cambios de posición de los elementos de la pantalla.

Inmediatamente después nos encontramos dos funciones directamente relacionadas con la interacción con el ratón. Se trata de las funciones `mousemuve` y `mousedown`.

La primera función, como la anterior, no devuelve nada pero sí requiere de dos enteros como argumentos. Éstos van referidos a las coordenadas x e y del ratón cuando éste se desplaza por la ventana. La estructura también es algo más compleja. Su dinamismo se basa directamente en el parámetro, antes mencionado, `controlmode`. Este parámetro únicamente puede tomar los valores 0, 1 ó 2, puesto que solamente hay 3 opciones de menú establecidas. El 0 determina el control de la cámara: cuando `controlmode` $= 0$, si desplazamos el cursor dentro de la pantalla que contiene a la escena, mientras mantenemos el botón izquierdo pulsado, podemos observar cómo la vista se desplaza alrededor del centro de ésta, de forma esférica, así, podemos observar lo que sucede desde diversos puntos de vista. El 1 determina el control sobre la esfera: si `controlmode` $= 1$, mientras que tengamos el botón pulsado, si desplazamos el ratón conseguiremos arrastrar la esfera por toda la escena (en el plano $y = 0$) y, de esta manera, poder interactuar con el paño. Por último, el 2 determina el control de la dirección del viento: siguiendo el mismo dinamismo que los casos anteriores, en el caso en el que `controlmode` $= 2$, podemos establecer la dirección del viento desde cualquier punto a lo largo del plano $y = 0$. La última acción de esta función es llamar a `idle` a fin de realizar la siguiente iteración de las posiciones de las partículas de la simulación con los nuevos datos proporcionados por los cambios anteriores.

La segunda únicamente se encarga de tomar las posiciones (x, y) del ratón en la pantalla con el fin de que, cuando pulsamos el bontón del ratón, iniciar desde dicho punto las interacciones descritas en la función anterior.

Al igual que el programa permite interactuar con el ratón, también lo podemos hacer con el teclado, y las posibilidades que nos ofrece de alterar el código nos las define la siguiente función.

Se trata de la función `keyboard`. Esta función se basa, principalmente, en el argumento `key` que requiere al ser llamada. Como se explica en el **Anexo 2**, a través de esta función se asigna a `key` un código ASCII que, según el valor que tome, tendrá un efecto u otro en nuestro programa. Estos efectos vienen determinados por la estructura por la cual está formada esta función. En ella se definen varios casos y toman papel los siguientes valores:

- **"esc"**.- Permite cerrar la ventana.
- **"+"** ó **"="**.- Aumenta en una unidad la velocidad del viento.
- **"-"**.- Disminuye en una unidad la velocidad del viento.
- **"f"**.- Inicializa el paño con el valor de `setuptype` = 1.
- **"c"**.- Inicializa el paño con el valor de `setuptype` = 0.
- **"z"**.- Aumenta la distancia del punto de vista al centro de la escena en 0.5 unidades.
- **"x"**.- Disminuye la distancia del punto de vista al centro de la escena en 0.5 unidades. Si la distancia es menor que 4, entonces la fija en 4.
- **"t"**.- Cambia el diseño de la textura: Aumenta en una unidad el parámetro `currenttexture` y la actualiza aplicándole módulo 4 a fin de tomar, únicamente, los valores 0, 1, 2 ó 3 para realizar, solamente, las 4 configuraciones de textura definidas más arriba.

Por último encontramos la función más importante de todas, la función `main` que es la que da inicio a la compilación y, la cual, es la primera en ser llamada en todo código de programación. En ella se inician las librerías de `GLUT` y la pantalla donde observaremos la escena de la simulación. Seguidamente, se crea el menú de accesos del ratón especificando qué se generará con el botón derecho de éste y, finalmente, se asocian las funciones específicas de `GLUT` a sus correspondientes en el código, declaradas anteriormente, de manera que permitan el correcto desarrollo del programa. La última de estas funciones es la función `MainLoop` que, como se explica en el **Anexo 2**, relaciona y pone en bucle las anteriores funciones para crear el dinamismo que podemos observar en la simulación al ejecutar el programa.

```
int main(int argc, char** argv) {
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
        glutInitWindowSize(800, 600);
        glutInitWindowPosition(100, 100);
        glutCreateWindow(argv[0]);
        init();

        //Setup menu
        glutCreateMenu(menu);
        glutAddMenuEntry("Mouse changes view direction", 0);
        glutAddMenuEntry("Mouse changes sphere position", 1);
        glutAddMenuEntry("Mouse changes wind direction", 2);
        glutAttachMenu(GLUT_RIGHT_BUTTON);

        //Set up control functions
        glutIdleFunc(idle);
        glutDisplayFunc(display);
        glutReshapeFunc(reshape);
        glutKeyboardFunc(keyboard);
        glutMotionFunc(mousemove);
        glutMouseFunc(mousedown);
        glutMainLoop();
        return 0;
}
```

2.4. Simplificación y mejora del código.

Llegados a este punto ya conocemos el programa en su totalidad, lo comprendemos y, por tanto, somos capaces de alterarlo. Al hacer el análisis tan exhaustivo que acabamos de hacer, hemos sido capaces de determinar algunas mejoras tanto para la visualización del programa como para la clarificación del código, a fin de poder trabajar con él de una forma más "matemática". A continuación detallaremos los cambios realizados y su justificación.

- En relación a la visualización hemos pasado de considerar 256 partículas a 10.000, tomando `CLOTH_RESOLUTION` = 100, consiguiendo que los pliegues que se producen en el paño en cada renderización no sean tan bruscos, y dé una mayor sensación de ductilidad.

- En cuanto a la eficacia del código, hemos cambiado algunos elementos que nos faciliten el poder interactuar con éste. En lo relacionado al ratón, hemos cambiado los mensajes emergentes por pantalla, que genera su menú, para que éstos aparezcan en español. Volviendo al control sobre la simulación, hemos añadido un parámetro booleano denominado `pause`, con la finalidad de poder congelar la simulación en cualquier momento que deseemos. Simplemente con pulsar la tecla "p", cambiamos el valor del boolean que, a través de un condicional situado en la función `idle`, actualizará o no la escena, y, con ello, congelará o descongelará la imagen.

  Así, también hemos hecho algunos cambios en cuanto a la limpieza del código. A lo largo del código podemos encontrar líneas de código repetidas o innecesarias. En primer lugar dentro de `initializecloth`, tras establecer las posiciones iniciales de las partículas, se copian en un segundo array, seguidamente se fijan los puntos extremos y, luego, se vuelve a realizar la copia de dichos arrays.

```
        for (i = 0; i < 4 * numparticles; i++)
                cloth2[i] = cloth1[i];

        //Initialize 2 corner points to infinite weight
        cloth1[3] = 0.0f;
        cloth1[4 * (CLOTH_RESOLUTION - 1) + 3] = 0.0f;

        for(i = 0; i < 4 * CLOTH_RESOLUTION * CLOTH_RESOLUTION; i++)
                cloth2[i] = cloth1[i];
```

  Esto es innecesario, bastará con fijar primeramente los puntos de sustentación del paño y realizar una única vez la copia de los arrays. En segundo lugar, es innecesario que dos teclas lleven la misma acción a la escena, por ello hemos eliminado la opción de que "=" pueda aumentar el viento. Tiene sentido que "+" aumente y "-" disminuya, el resto es dispensable. Y, finalmente, con el objetivo de clarificar las lineas del código, hemos hecho comentarios en todos los lugares que hemos considerado oportunos, destacando la función `calculatenormals` donde explicamos, mediante diagramas, qué vectores utilizamos para el cálculo de cada normal.

- Por último hemos realizado algunos cambios directamente sobre la programación en cuanto a la forma de la implementación del método numérico. En matemáticas, a la hora de programar métodos nos basamos en su estructura analítica. En general, dicha estructura viene dada por la forma (1.1) donde, y_{n+1} se calcula a partir de una función $f(t, y(t))$ dada. De modo que, en nuestro problema, lo que buscamos es tener este tipo de estructura que nos ayudará a organizarnos mejor. En el código, hay definida la función `accumulateforces` que actualiza, de manera externa a la ecuación, los valores de la función $f(t, y(t))$ que más tarde utilizará ésta. Desde un enfoque matemático, esto no tiene sentido, por lo que hemos

desarrollado e implementado una función `f` que sustituirá la anterior y, la cual, calculará los valores de la pendiente de forma dependiente a los parámetros que interfieren en el cálculo, y no de forma ajena como antes.

```
float f(float t, float yAnt, float yAct, char var, float fuerzaViento, float
    fuerzaRozamiento, float normal, float masa) {
        float coef = 80.0;

        float funcion;
        if (var == 'x' || var == 'z') {
                funcion = 0.0;
        }
        else if (var == 'y') {
                funcion = gravityforce;
        }

        funcion += fuerzaViento * normal + fuerzaRozamiento * normal * coef;
        funcion = funcion / masa;

        return  funcion;
}
```

Así, dicha función permitirá establecer, de forma más genérica, un código global que pueda abarcar diferentes ecuaciones. Puesto que nuestro sistema depende de fuerzas más globales como la del viento o la de rozamiento, y el método se aplica sobre cada coordenada, establecemos el código de manera que podamos calcular, dentro de nuestro `verlet`, dicho vector director y pasar los datos necesarios para el cálculo de la pendiente como parámetros, además de los propios: tiempo y posición actuales. A parte de asociar a nuestra función `verlet` la capacidad del cálculo de la pendiente (mediante la anterior función externa `f`), es ésta la encargada de llamar a la actualización de las normales que será información relevante para dicho cálculo. Así, y cambiando de posición en el código la función `calculatenormals`, a fin de no tener que realizar una función prototipo, podemos reajustar el papel de la función `updatecloth` y que sea la función `idle` la que asuma su papel. De esta manera eliminamos, así, una función que, tras los cambios mencionados, había perdido toda utilidad.

Finalmente, y como resultado, obtenemos un código elaborado que nos será de mayor utilidad que el inicial a la hora de plantear los diferentes cambios que queremos llevar a cabo.

2.5. Reducción a una partícula. Comprobación del orden.

Lo que hemos conseguido con la sección anterior es tener un código con el que nos sintamos más cómodos trabajando ya que queda adecuado a la manera usual en la que nosotros, como programadores, estamos habituados a trabajar.

Ahora, lo que haremos en primer lugar, será simplificar el código hasta el punto de ser capaces de dibujar, únicamente, una partícula en la escena. Para ser conscientes de la posición relativa de ésta en el entorno visual, dibujaremos los ejes coordenados de fondo. Una vez hecho esto, intentaremos darle movimiento a través de métodos numéricos de un paso. Lo que haremos será trabajar con ecuaciones más simples a fin de que podamos comparar la solución aproximada, obtenida con el método, con la exacta; y comprobar que, efectivamente, el orden de aproximación del método es correcto.

2.5.1. Dibujar una partícula.

Aunque, en el código de la sección anterior podemos observar funciones realmente complejas, para entender cómo funciona C, y una primera aproximación a la librería `GLUT` de OpenGL, hemos simplificado la programación hasta el nivel más básico. Como hemos mencionado: nuestro objetivo era ser capaces de dibujar un punto en una escena de 3 dimensiones con unos ejes centrados que nos ayudaran a tener una referencia del espacio mostrado. El resultado es el que podemos observar en la **Figura 2.2**

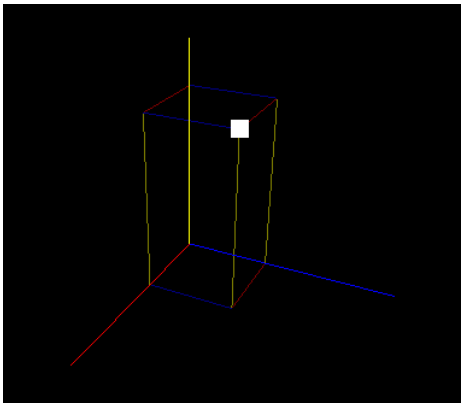


Figura 2.2: Dibujo de una partícula en 3D.

Lo que hemos hecho ha sido eliminar todas las funciones relacionadas con los métodos numéricos, ya que en una primera instancia la partícula no se va a mover, así como las encargadas de la iluminación, las interacciones por pantalla con ratón y teclado, o aquellas que no tenían un papel significativo. De este modo nuestra función `main` queda reducida al siguiente código:

```
int main(int argc, char** argv) {
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
        glutInitWindowSize(850, 600);
        glutInitWindowPosition(100, 50);
        glutCreateWindow(argv[0]);
        glPointSize(15.0);
        initializecloth();

        //Organizacion de las funciones de control
        glutDisplayFunc(display);
        glutReshapeFunc(reshape);
        glutMainLoop();
        return 0;
}
```

Aquí únicamente definimos la pantalla, inicializamos los elementos que forman parte de nuestro sistema y llamamos a las funciones básicas para la renderización. Por otra parte, para realizar el dibujo, hemos redeclarado nuestra función `drawcloth`. De esta manera, señalando nosotros el punto que deseamos dibujar y algunas líneas para observar la sensación de 3D, declaramos el entorno de dibujo entre los comandos `glBegin` y `glEnd` especificando, al mismo tiempo, el tipo de primitiva que queremos dibujar y, a través de las llamadas:

```
glBegin(GL_POINTS);
        glColor3f(1.0f, 1.0f, 1.0f);
        glVertex3f(currentcloth[0], currentcloth[1], currentcloth[2]);
glEnd();
```

para el punto y

```
glBegin(GL_LINES);
        //Lineas azul
        glColor3f(0.5f, 0.5f, 0.0f);
        glVertex3f(x_0, 0.0, 0.0);
        glVertex3f(x_0, y_0, 0.0);

        .
        .
        .
glEnd();
```

para las líneas, obtenemos el ya citado resultado de la **Figura 2.2**.

2.5.2. Evolución hasta el 3D

El haber conseguido dibujar una partícula de manera estática en la escena nos ha permitido adquirir una gran desenvoltura en cuanto a la dinamización de la impresión de objetos por pantalla. Ahora, nos gustaría ir avanzando poco a poco hacia la realidad virtual del programa original, es decir, dar movimiento a la partícula en cada una de las dimensiones de la pantalla.

En una primera instancia nos redujimos al caso más básico: trabajamos sobre una dimensión y realizamos un ejemplo sencillo basado en el método de Euler, ya mencionado en la página 3. Dicho ejemplo resulta del estudio de la caída libre de una partícula. De esta manera, tras un proceso de adaptación del código, declaramos el método de Euler para dinamizar el método, dando como resultado la **Figura 2.3**

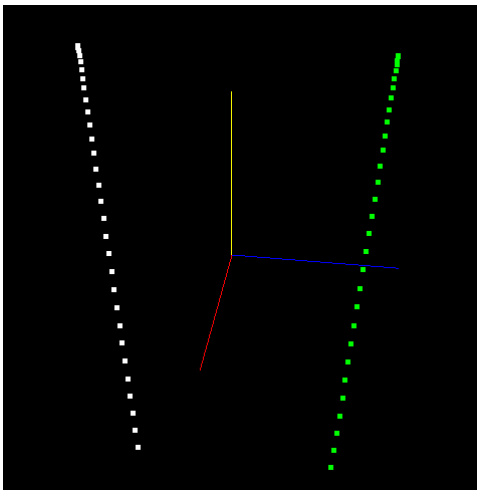


Figura 2.3: Trayectoria de caída libre para la partícula. A la izquierda se puede observar el trazo de la aproximada y a la derecha la solución exacta, para evitar superposiciones.

El siguiente paso fue elaborar el método de RK2 para realizar otro ejemplo, ahora con dos dimensiones. Basándonos de nuevo en un ejemplo básico como lo es un movimiento armónico simple, implementamos en el código dicho método a partir de la estructura (2.2), con las correspondientes funciones $f(t_n, y_n, v_n)$ y $g(t_n, y_n, v_n)$ para y' y v', respectivamente, dando lugar a la **Figura 2.4**

Finalmente, ya nos encontramos en disposición de poder dar el paso definitivo: trabajar con una partícula que se desplace en las tres dimensiones de la escena. Por ello, un ejemplo sencillo y que tiene un cierto atractivo es el de una partícula cuya trayectoria define un movimiento

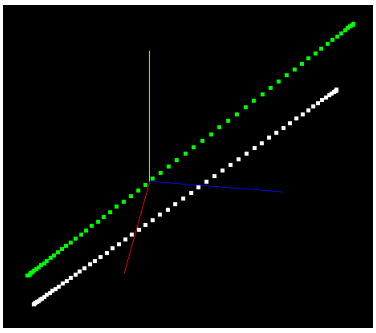


Figura 2.4: Trayectoria del M.A.S. para la partícula. En la parte inferior, aproximación y en la superior, solución exacta.

helicoidal, originando así, visualmente, una hélice. Para este ejemplo implementamos otro de los métodos de la familia Runge-Kutta, en este caso RK4. De esta manera obtenemos los resultados de la **Figura 2.5**

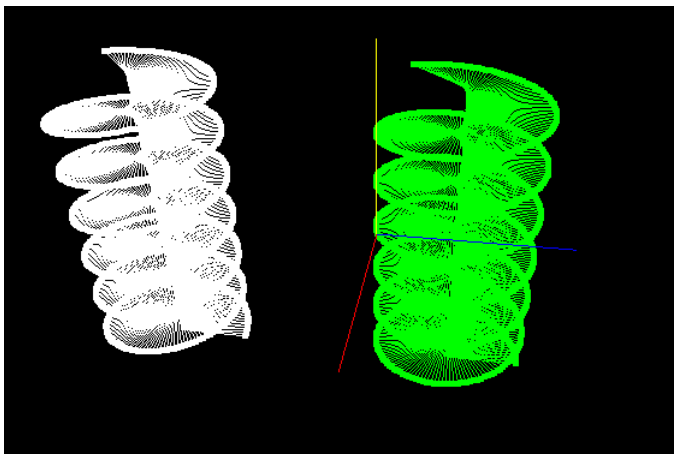


Figura 2.5: Trayectoria de hélice para la partícula. A la izquierda, aproximación y a la derecha, solución exacta.

2.5.3. Comprobación del Orden

A lo largo de la sección anterior hemos ido desarrollando distintos ejemplos que nos han permitido avanzar en las distintas dimensiones de la "realidad", al mismo tiempo que también evolucionábamos con métodos numéricos más elaborados. Sin embargo, el avanzar como hemos hecho utilizando dichos métodos se queda un poco vacío si esto no condujera a un objetivo final. En el Capítulo 1 pudimos comprobar cómo, para problemas rígidos, desarrollar un método numérico más complejo, como los métodos adaptativos, nos permitía obtener resultados más próximos a la solución exacta del problema. No obstante, eso no siempre es lo más eficaz. En los códigos que hemos desarrollado en este capítulo, implementar un método adaptativo, además de no ser sencillo, lleva una carga computacional que podría ralentizar enormemente la visualización dinámica del programa. Sin embargo, esto no quiere decir que no podamos conseguir buenas aproximaciones a través de otras técnicas diferentes. Es por ello que al igual que con cada ejemplo avanzábamos en una dimensión el problema, también avanzábamos con el orden del método numérico utilizado. Bien es sabido que, cuanto mayor sea el orden de un método, mayor será la rapidez de aproximación a la solución exacta respecto del paso de tiempo.

Para justificar esta aproximación y comprobar que el orden de los métodos elaborados es, efectivamente, el que debe ser y, por tanto, que nuestros métodos están bien construidos procederemos a hacer un estudio que nos aporte esta información de una forma más gráfica y visual. Puesto que los ejemplos desarrollados dependen únicamente del tiempo en cada variable, recurriremos al libro Kincaid-Cheney [8, pág 489] donde podemos encontrar un buen ejemplo de sistema de ecuaciones, $\vec{x}'(t) = A\vec{x}(t) + \vec{C}(t)$, en tres dimensiones con $\vec{C}$ constante.

En este caso, lo que haremos será utilizar los métodos que hemos desarrollado en los ejemplos anteriores para resolver el sistema que se plantea. Realizaremos la simulación para cada método un número M de veces, con M pasos diferentes, de forma que, si N_i es la partición para la etapa i, entonces $N_{i+1} = 2N_i$ con $i = 1 \dots M$. Así, podremos calcular los errores obtenidos con cada partición y guardarlos en un fichero externo que podamos leer posteriormente con MATLAB. Con estos ficheros elaboraremos una tabla de errores y particiones, con la que podremos realizar una "gráfica de errores" construyendo, en la recta de abscisas, $-\log(N_i)$, y de ordenadas, su correspondiente error. Comparando la recta obtenidas con rectas de pendientes uno, dos y cuatro, podremos determinar, con exactitud, el orden del método y corroborar que el método funciona correctamente con el orden que se espera.

Así, presentamos el problema en cuestión a tratar. Dicho problema consta de una matriz

$$A = \begin{pmatrix} -8/3 & -4/3 & 1 \\ -17/3 & -4/3 & 1 \\ -35/3 & -4/3 & -2 \end{pmatrix}$$

y un vector

$$\vec{C}(t) = \begin{pmatrix} 12 \\ 29 \\ 48 \end{pmatrix}$$

de forma que las soluciones exactas del sistema vienen dadas por:

$$\begin{cases} x(t) = \frac{1}{6}e^{-3t}(6 - 50e^{t} + 10e^{2t} + 34e^{3t}) \\ y(t) = \frac{1}{6}e^{-3t}(12 - 125e^{t} + 40e^{2t} + 73e^{3t}) \\ z(t) = \frac{1}{6}e^{-3t}(14 - 200e^{t} + 70e^{2t} + 116e^{3t}) \end{cases} \tag{2.3}$$

Teniendo estos datos, implementamos nuestros métodos numéricos en el código, de manera que tenemos nuestra función `EulerExplicito` dada por

```
void EulerExplicito(void) {
        float *auxPos;

        previouscloth[0] = currentcloth[0] + TIME_STEP * f(simulationtime, currentcloth
            [0], currentcloth[1], currentcloth[2], 'x');
        previouscloth[1] = currentcloth[1] + TIME_STEP * f(simulationtime, currentcloth
            [0], currentcloth[1], currentcloth[2], 'y');
        previouscloth[2] = currentcloth[2] + TIME_STEP * f(simulationtime, currentcloth
            [0], currentcloth[1], currentcloth[2], 'z');

        auxPos = previouscloth;
        previouscloth = currentcloth;
        currentcloth = auxPos;

        //exacta
        exacta(simulationtime);
}
```

nuestra función `RungeKutta2` por

```
void RungeKutta2(void) {
        float *auxPos;
        char variable[] = { 'x', 'y', 'z' };
        float k1[3], k2[3];

        for (int i = 0; i < 3; i++) {
                k1[i] = f(simulationtime, currentcloth[0], currentcloth[1], currentcloth
                    [2], variable[i]);
        }
        for (int i = 0; i < 3; i++) {
                k2[i] = f(simulationtime + TIME_STEP, currentcloth[0] + TIME_STEP * k1
                    [0], currentcloth[1] + TIME_STEP * k1[1],
                        currentcloth[2] + TIME_STEP * k1[2], variable[i]);
        }

        float coef = TIME_STEP / 2.0;
        previouscloth[0] = currentcloth[0] + coef * (k1[0] + k2[0]);
        previouscloth[1] = currentcloth[1] + coef * (k1[1] + k2[1]);
        previouscloth[2] = currentcloth[2] + coef * (k1[2] + k2[2]);

        //Actualizacion de variables
        auxPos = previouscloth;
        previouscloth = currentcloth;
        currentcloth = auxPos;

        //exacta
        float tiempo = simulationtime + TIME_STEP;
        exacta(tiempo);
}
```

y, nuestro `RungeKutta4` por

```
void RungeKutta4(void) {
        float *auxPos;
        char variable[] = { 'x', 'y', 'z' };
        float k1[3], k2[3], k3[3], k4[3];

        for (int i = 0; i < 3; i++) {
                k1[i] = f(simulationtime, currentcloth[0], currentcloth[1], currentcloth
                    [2], variable[i]);
        }
        for (int i = 0; i < 3; i++) {
                k2[i] = f(simulationtime + TIME_STEP / 2.0, currentcloth[0] + TIME_STEP /
                     2.0 * k1[0], currentcloth[1] + TIME_STEP / 2.0 * k1[1],
                        currentcloth[2] + TIME_STEP / 2.0 * k1[2], variable[i]);
        }
        for (int i = 0; i < 3; i++) {
                k3[i] = f(simulationtime + TIME_STEP / 2.0, currentcloth[0] + TIME_STEP /
                     2.0 * k2[0], currentcloth[1] + TIME_STEP / 2.0 * k2[1],
                        currentcloth[2] + TIME_STEP / 2.0 * k2[2], variable[i]);
        }
        for (int i = 0; i < 3; i++) {
                k4[i] = f(simulationtime + TIME_STEP, currentcloth[0] + TIME_STEP * k3
                    [0], currentcloth[1] + TIME_STEP * k3[1],
                        currentcloth[2] + TIME_STEP * k3[2], variable[i]);
        }

        float coef = TIME_STEP / 6.0;
        previouscloth[0] = currentcloth[0] + coef * (k1[0] + 2.0 * k2[0] + 2.0 * k3[0] +
            k4[0]);
        previouscloth[1] = currentcloth[1] + coef * (k1[1] + 2.0 * k2[1] + 2.0 * k3[1] +
            k4[1]);
        previouscloth[2] = currentcloth[2] + coef * (k1[2] + 2.0 * k2[2] + 2.0 * k3[2] +
            k4[2]);

        auxPos = previouscloth;
        previouscloth = currentcloth;
        currentcloth = auxPos;

        //exacta
```

```
        float tiempo = simulationtime + TIME_STEP;
        exacta(tiempo);
}
```

De igual modo, implementamos nuestra función `f`, por la que se rigen los cálculos de nuestros métodos. Resaltemos que, dado que los métodos Runge-Kutta necesitan de puntos medios auxiliares entre los pasos establecidos, a fin de realizar su promedio de pendientes, en lugar de pasar el vector entero de datos como parámetro, se pasarán sus coordenadas para facilitar los cálculos de las distintas pendientes de paso. De esta forma, la función `f` queda declarada como sigue

```
float f(float t, float x, float y, float z, char var) {
        float funcion;
        if (var == 'x') {
                funcion = - 8.0/3.0 * x - 4.0/3.0 * y + z + 12;
        } else if (var == 'y') {
                funcion = -17.0 / 3.0 * x - 4.0 / 3.0 * y + z + 29;
        } else if (var == 'z') {
                funcion = -35.0 / 3.0 * x + 14.0 / 3.0 * y - 2 * z + 48;
        }

        return  funcion;
}
```

al igual que las coordenadas de nuestra función exacta, que, como bien se indica en (2.3) viene declarada como

```
void exacta(float t) {
        yExacta[0] = 1.0 / 6.0 * exp(-3 * t) * (6 - 50 * exp(t) + 10 * exp(2 * t) + 34 *
            exp(3 * t));
        yExacta[1] = 1.0 / 6.0 * exp(-3 * t) * (12 - 125 * exp(t) + 40 * exp(2 * t) + 73
            * exp(3 * t));
        yExacta[2] = 1.0 / 6.0 * exp(-3 * t) * (14 - 200 * exp(t) + 70 * exp(2 * t) + 116
             * exp(3 * t));
}
```

Procediendo de este modo, obtenemos el resultado visual de la **Figura 2.6** donde, como podemos ver, tanto la aproximación como la trayectoria exacta prácticamente coinciden.

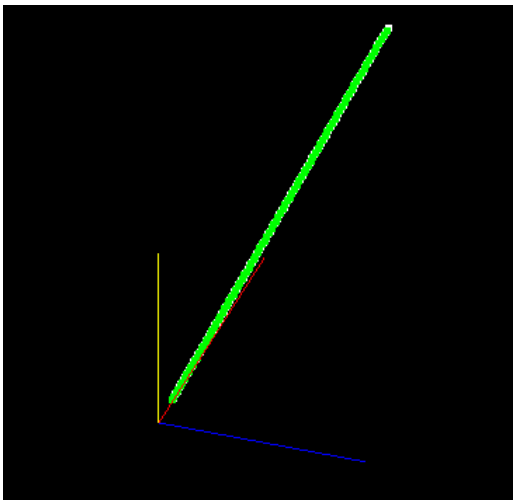


Figura 2.6: Resultado gráfico en 3D del ejemplo propuesto.

Realizando los ficheros tal y como hemos especificado anteriormente, obtenemos las gráficas de errores realizadas con MATLAB. En la **Figura 2.7** podemos observar cómo, la gráfica origi-

nada por los logaritmos de los errores y las particiones da lugar a una recta de pendiente uno, como cabía esperar, reflejando así el orden de este método.

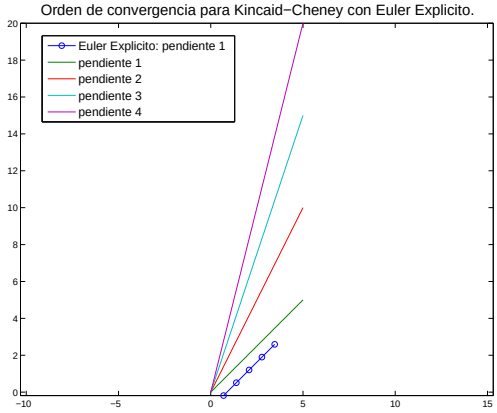


Figura 2.7: Orden de convergencia de Euler Explícito: recta generada de pendiente uno.

De igual modo se generan las gráficas en la **Figura 2.8** para Runge-Kutta 2, obteniendo, del mismo modo, una recta de pendiente dos que respalda el orden del método.

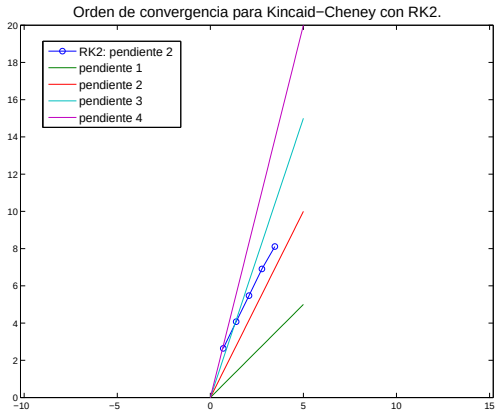


Figura 2.8: Orden de convergencia de RK2: recta generada de pendiente dos.

Y, finalmente, con la misma comprobación que las anteriores, podemos determinar una conclusión análoga para el método de Runge-Kutta 4 con la **Figura 2.9**

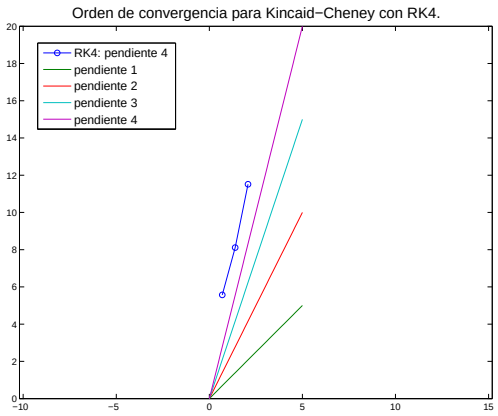


Figura 2.9: Orden de convergencia de RK4: recta generada de pendiente cuatro.

2.5.4. Ejemplo dinámico: Atractor de Rössel y Lorenz

Habiendo ya justificado que los métodos que utilizamos, efectivamente, están funcionando correctamente y que su resultado es del orden esperado, podemos atrevernos a realizar computaciones más elaboradas. Para ellos recurriremos a la resolución de las ecuaciones de dos de los atractores más famosos de las matemáticas: el atractor de Rössler[4] y el de Lorenz[5].

Dichos atractores se basan en sistemas de ecuaciones, en este caso no lineales, de sistemas caóticos. Así, la órbita que recorren las soluciones de estos dos sistemas genera dos figuras en 3D que, además de atractivas visualmente, son realmente curiosas. En su estudio únicamente nos centraremos en la parte correspondiente al cálculo y la obtención de su solución, sin entrar de un modo más profundo en lo concerniente a la teoría del caos, y sus propiedades ante ésta. Simplemente nos servirán de ejemplos gráficos y visuales. Para ello acudiremos al informe [5], de donde hemos sacado los datos correspondientes a los ejemplos.

Atractor de Rössler

Presentemos, en primer lugar, en el atractor de Rössler. Las ecuaciones que definen un sistema de Rössler vienen determinadas por el siguiente sistema de ecuaciones diferenciales:

$$\begin{cases} x'(t) = -y(t) - z(t) \\ y'(t) = x(t) + ay(t) \\ z'(t) = b + z(t)(x(t) - c) \end{cases} \tag{2.4}$$

donde a, b, c son parámetros reales. Dichas ecuaciones modelizan la "reacción química de Belousov-Zhabotinsky" que es la oxidación de malonato mediante bromato en presencia de iones metálicos.

Tomando así las condiciones iniciales $x(0) = y(0) = z(0) = 1.0$, como parámetros $a = b = 0.2$ y $c = 5.7$, con una partición de $N = 100000$ y un tiempo total de $T = 1000$; podemos declarar una función `f` en nuestro código que contenga estas ecuaciones obteniendo resultados como el de la **Figura 2.10**.

[4]Otto Eberhard Rössler es un bioquímico conocido por su trabajo en la teoría del caos y su ecuación teórica conocida como atractor de Rössler. Más información en https://en.wikipedia.org/wiki/Otto_Rossler

[5]Edward Norton Lorenz fue un matemático y metereólogo americano pionero en la teoría del caos. Fue él quien introdujo la noción de "atractor extraño" y acuñó el término de "efecto mariposa". Más información en https://en.wikipedia.org/wiki/Edward_Norton_Lorenz

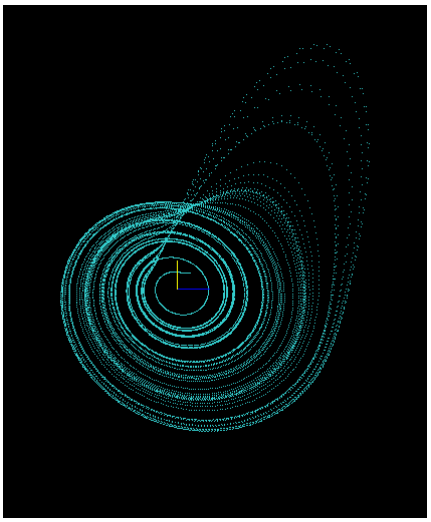


Figura 2.10: Atractor de Rössler tomando como valores de los parámetros $a = b = 0.2$ y $c = 5.7$

Utilizando un fichero externo que nos guarde las coordenadas de esta simulación podemos comprobar, con MATLAB, las magnitudes de la gráfica en tres dimensiones en la figura **Figura 2.11**, y sus proyecciones sobre el plano z y sobre el plano y, en la **Figura 2.12**.

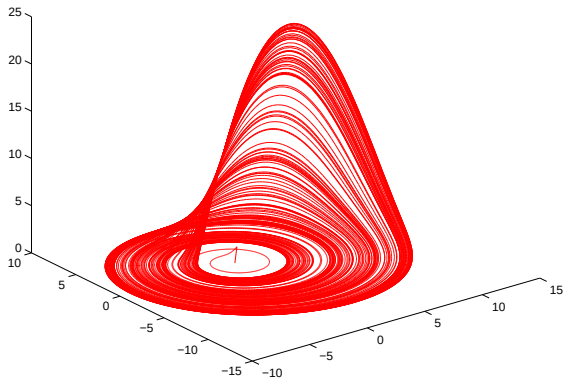


Figura 2.11: Atractor de Rössler con los datos de fichero en MATLAB.

Atractor de Lorenz

Por otro lado, tenemos también el atractor de Lorenz, cuyas ecuaciones vienen descritas por el sistema

$$\begin{cases} x'(t) = \sigma(y(t) - x(t)) \\ y'(t) = x(t)(\rho - z(t)) - y(t) \\ z'(t) = x(t)y(t) - bz(t) \end{cases} \tag{2.5}$$

donde destacan σ, que es el número de Prandtl[6], y ρ, parámetro de control proporcional al incremento de T. Dichas ecuaciones pretenden realizar una descripción realista de la convección

[6]Número adimensional proporcional al cociente entre la viscosidad y la difusión térmica.

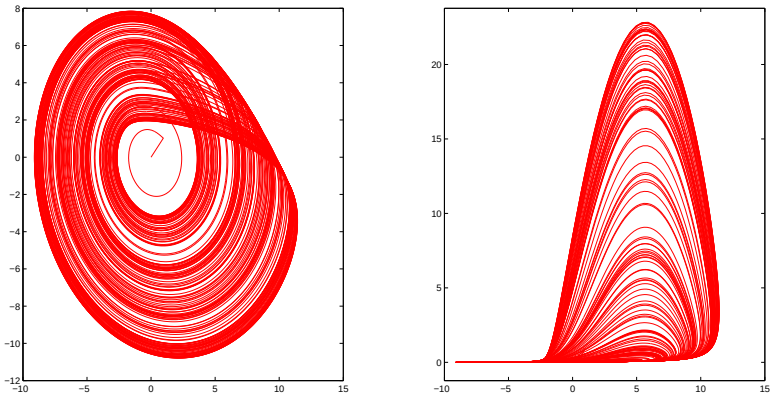


Figura 2.12: Proyecciones del atractor de Rössler sobre los ejes xy (izquierda) y sobre los ejes xz (derecha).

atmosférica. Son el resultado de una simplificación drástica del problema inicial, pero da lugar a uno de los avances más importantes en el estudio de los sistemas dinámicos. Sus coordenadas representan la velocidad y la temperatura del fluido.

Tomando las mismas condiciones iniciales que en el ejemplo anterior y calculando la órbita que originan en un intervalo de tiempo $T = 100$ con la misma partición, podemos observar en la **Figura 2.13** el resultado en 3D.

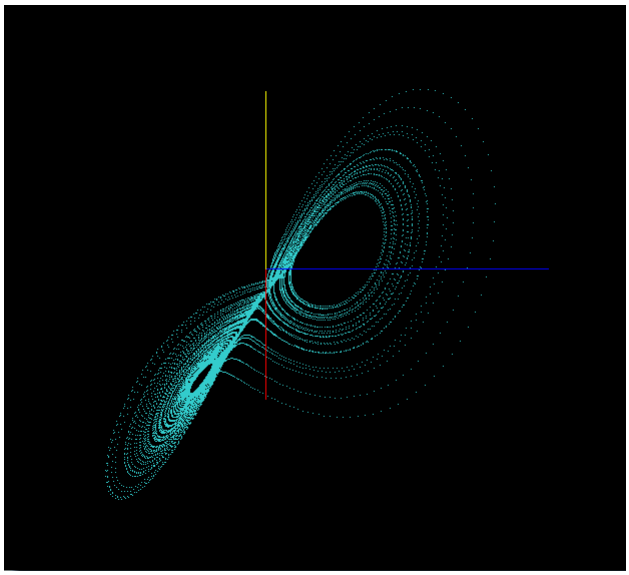


Figura 2.13: Atractor de Lorenz tomando como valores de los parámetros $\sigma = 10$, $\rho = 28$, y $b = 8/3$

Al igual que en el ejemplo anterior, elaboramos un fichero externo que podamos leer con MATLAB a fin de visualizar la imagen en 3D, **Figura 2.14**, y, aprovechando los datos, también sus proyecciones sobre los ejes xy y xz, dados en la **Figura 2.15**

De esta manera finalizamos así, un apartado en el que hemos podido llevar a cabo unos ejemplos, fuera de la linealidad original con la que empezamos a trabajar, y que permiten la elaboración de llamativas figuras en 3D.

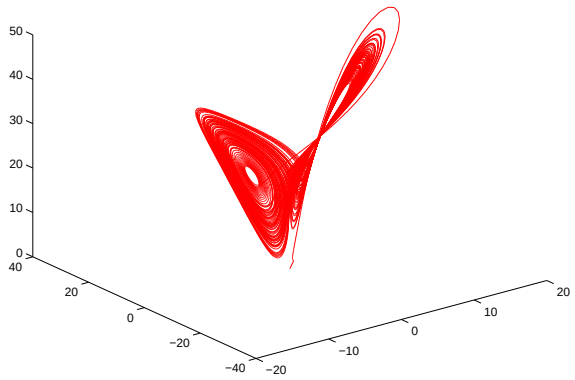


Figura 2.14: Atractor de Lorenz con los datos de fichero en MATLAB.

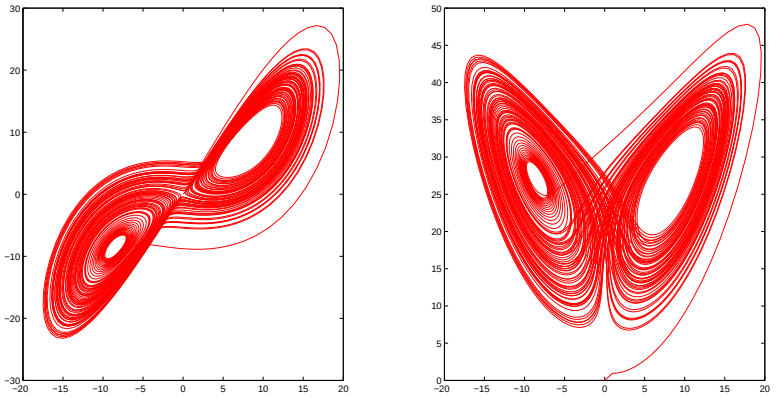


Figura 2.15: Proyecciones del atractor de Lorenz sobre los ejes xy (izquierda) y sobre los ejes xz (derecha).

2.6. Ley de Hooke

Reflejada ya la soltura que hemos adquirido con el código, nos gustaría retomar el programa original con fin de resolver una de las cuestiones que no se quedan, a nuestro juicio, debidamente planteadas en éste. Uno de los puntos destacables de toda simulación, que conlleve digitalizar cualquier tejido, es la capacidad de darle consistencia al material, de manera que se quede unido y dé la apariencia adecuada. Como mencionamos en la página 27, al hablar sobre la función `satisfayconstraints`, este código resuelve este problema corrigiendo, tras cada iteración del método numérico, la posición de las partículas que conforman la tela de forma "manual", sin ninguna física que lo apoye, simplemente como corrección de la posición. Además, dicho procedimiento se realiza varias veces lo cual, aunque visualmente aporte una "realidad" evidente, no es la forma correcta de proceder.

Lo que nosotros vamos a hacer en esta sección será intentar corregir este planteamiento. Como ya citamos en la página anteriormente referida, a la hora de intentar dar una "consistencia digital" al simular tejidos virtualmente, lo que se propone es materializarlo a través de un conjunto, relativamente grande, de partículas. A fin de que estas partículas tengan una dependencia unas de otras para dar efecto de un sólo elemento, nuestra prenda, se establece una fuerza de

atracción entre ellas, simulando la atracción entre las partículas de cualquier tejido en la vida real. Esta "fuerza de atracción" entre las partículas se imita proponiendo la existencia de un muelle que las una y aplicando a éste la Ley de Hooke.

2.6.1. Muelle fijo. Sistema con una única partícula

Procediendo de esta manera, plantearemos en primer lugar un ejemplo sencillo a partir del artículo de Hauth y Etzmuss [12] que podemos encontrar en la sección 3.2. En él, simularemos la acción de un muelle fijado a una superficie que tiene, en su otro extremo, anclada una partícula de masa m. Dicha partícula se moverá viéndose afectada por la fuerza proporcionada por el muelle y la gravedad del sistema llegando, finalmente, a un punto de equilibrio estático. La ecuación que describe esta trayectoria viene dada de la siguiente forma:

$$\frac{d^2z}{dt^2} = \frac{k}{m}(l_0 - z) - \frac{d}{m}\frac{dz}{dt} + g_z$$

que, expresado en nuestra notación, y adecuada a la disposición de los ejes coordenados con los que trabajamos, queda:

$$y''(t) = -\frac{k}{m}(y - L) - \frac{d}{m}y'(t) + g \tag{2.6}$$

donde $y(t)$, como en los otros casos, denota la posición de la partícula en el instante t, k establece la rigidez del muelle, L la longitud en reposo de éste, d es la constante que indica el rozamiento de la partícula con el medio, y g es la fuerza de gravedad del sistema.

En estas condiciones cabe destacar la dependencia que existe entre k y d. Haciendo referencia al libro de Física Universitaria, [17, pág 441], podemos diferenciar 3 casos fundamentales:

- $\boxed{d = 2\sqrt{km}} \Longrightarrow$ Amortiguamiento Crítico. El sistema no oscila, vuelve a la posición de equilibrio sin oscilar cuando se desplaza y se suelta.
- $\boxed{d > 2\sqrt{km}} \Longrightarrow$ Sobreamortiguamiento. No hay oscilación, el sistema regresa al equilibrio más lentamente que con amortiguamiento crítico. En este caso la solución de (2.6) viene dada por
  $$y(t) = C_1e^{-a_1t} + C_2e^{-a_2t} + C_3$$
  donde C_1, C_2 y C_3 dependen de las condiciones iniciales y, a_1 y a_2, dependen de m, k y d.
- $\boxed{d < 2\sqrt{km}} \Longrightarrow$ Subamortiguamiento. En este caso, el sistema oscila con amplitud constantemente decreciente. Éste será el caso de nuestro estudio.

Basándonos en los datos que nos aporta el artículo antes citado, las condiciones de nuestro problema serán: $m = 1$, $k = 1000$, $L = -1$, $d = 10$, $g = -10$, $v_0 = -5$ e $y_0 = -2$, los cuales corresponden al caso del "subamortiguamiento". Puesto que la ecuación (2.6) nos ofrece una EDO de 2 orden basada en la fuerza, el procedimiento para los métodos numéricos será el mismo que hemos seguido hasta ahora, a través del sistema (2.2), y aplicando los métodos ya desarrollados en el capítulo.

Partiendo de la ecuación (2.6), y trabajando un poco previamente a fin de dejarla en su forma estándar con coeficientes constantes, realizando sencillos pasos aritméticos, llegamos a

$$y''(t) + Dy'(t) + Ky(t) = C \tag{2.7}$$

donde $D = \frac{d}{m}$, $K = \frac{k}{m}$ y $C = \frac{k}{m} + g$.

Llegados a este punto, podemos determinar los valores propios del polinomio característico de esta ecuación. Realizando sencillos pasos obtenemos que dichos autovalores vienen dados por

$$\lambda_{1,2} = \frac{-D \pm \sqrt{D^2 - 4K}}{2}$$

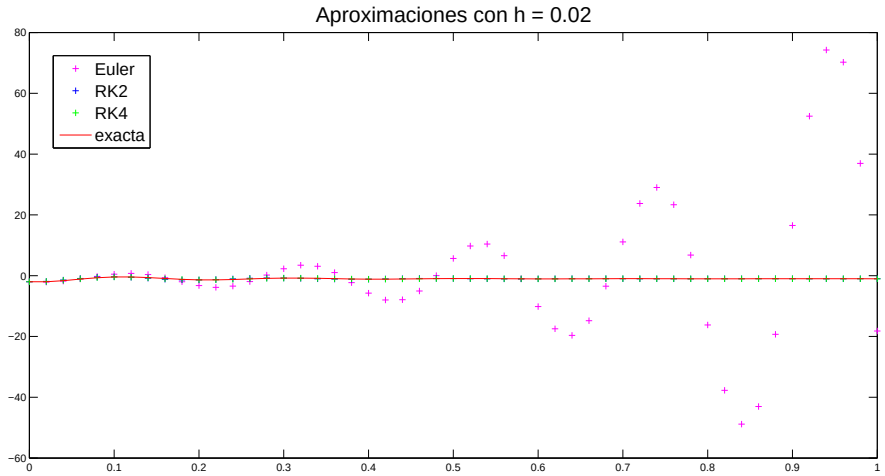
Y dado que nos encontramos en el caso subamortiguado, la solución son dos raíces complejas conjugadas, es decir,

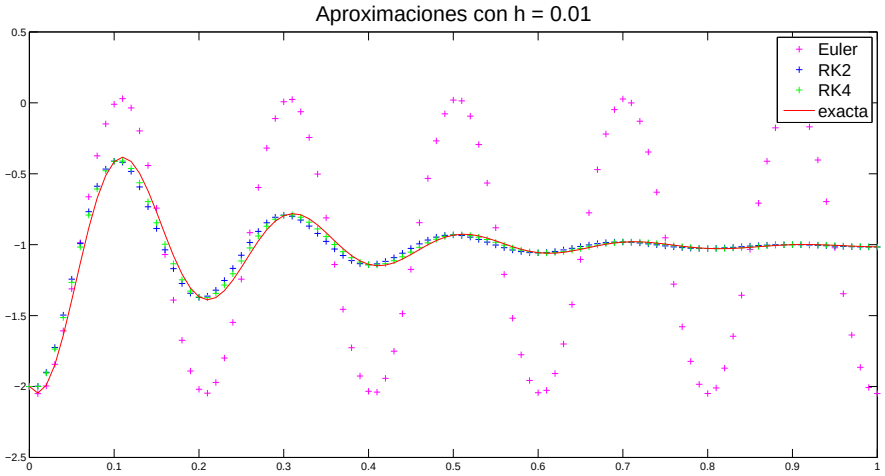
$$\begin{cases} \lambda_1 = -\frac{D}{2} + \frac{\sqrt{4K-D^2}}{2}\imath \\ \lambda_2 = -\frac{D}{2} - \frac{\sqrt{4K-D^2}}{2}\imath \end{cases}$$

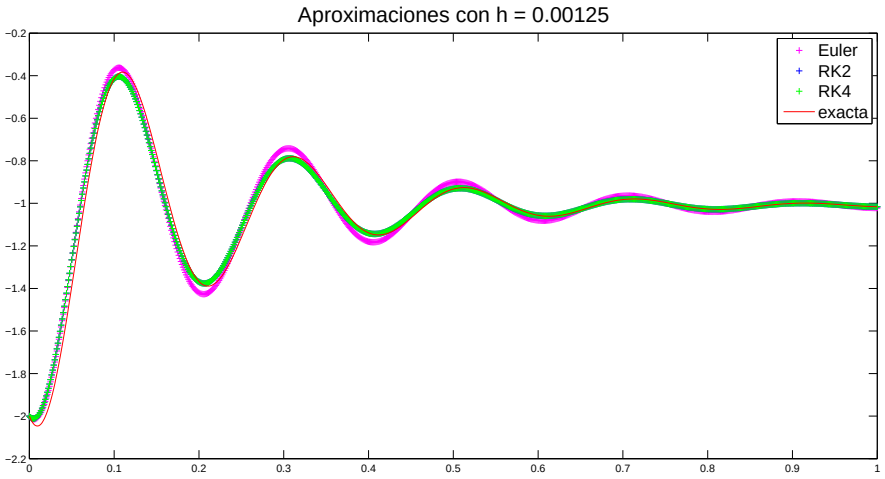
De este modo podemos concluir que la solución exacta de la ecuación del muelle, para nuestro ejemplo, viene dada por la expresión

$$y(t) = \frac{C}{K} + y_0 e^{-\frac{D}{2}t} \cos\left(\frac{\sqrt{4K - D^2}}{2}t\right) + \frac{2v_0 + Dy_0}{\sqrt{4K - D^2}} e^{-\frac{D}{2}t} \operatorname{sen}\left(\frac{\sqrt{4K - D^2}}{2}t\right) \tag{2.8}$$

Teniendo ahora la ecuación exacta y los esquemas numéricos, podemos ya realizar una comparación de los mismos con los diferentes pasos indicados en el artículo. Procediendo de esta manera obtenemos los siguientes resultados para los distintos valores del paso h:







tomando como datos

```
%Datos
%Constante de MUELLE
m = 1.0;
L = -1.0;
k = 1000.0;
d = 10.0;
gravity = -10.0;
y0 = -2.0;
v0 = -5.0;

%Constantes derivadas
K = k / m;
D = d / m;
raiz = sqrt(4 * K - D*D);
C = L * K + gravity;
C1 = y0 - C / K;
C2 = (2 * v0 + y0 * D) / raiz;

T = 1.0;
N = 800; %       50  | 100  |   800
h = T/N; %  0.02 | 0.01 | 0.00125
```

como función exacta

```
yex = C1 * exp(-D/2*t).*cos(raiz/2*t) + C2*exp(-D/2*t).*sin(raiz/2*t) + C / K ;
```

y como funciones `f` y `g`:

```
function z=f(tt,yy,vv)
    z = vv;
end
```

```
function z=g(tt,yy,vv)
    z = - K * (yy - L) - D * vv + gravity;
end
```

En estos resultados llevados a cabo con MATLAB, podemos observar la clara diferencia que existe entre la estabilidad que presentan cada uno de los métodos comparados, respecto al paso de tiempo utilizado. Como cabe esperar, los métodos Runge-Kutta aproximan mucho mejor

la ecuación con paso de tiempo más grande, mientras que, para Euler Explícito, es necesario reducir enormemente el paso a fin de que la aproximación sea estable y correcta. Evitando así que explote el sistema y los resultados obtenidos carezcan de sentido.

2.6.2. Sistema de Muelles

Una vez hemos desarrollado la teoría sobre la fuerza de Hooke, comprobando aproximación y solución exacta de la trayectoria de una partícula sometida a tal fuerza, nos disponemos a generalizar este caso. Nuestro objetivo va a ser establecer un sistema de partícula de forma que entre cada dos partículas adyacentes exista un muelle que las una, de esta manera, generar una fuerza de atracción en cadena entre todas ellas a fin de originar una malla sostenida de dos de sus extremos y que cuelgue a modo de "sábana tendida".

Para ello, basándonos en los artículos [15, pág 3] y [12, pág 3], pasaremos la ecuación (2.6) a un sistema de forma que intervengan todas las partículas. Procediendo de esta manera obtenemos que, para cada partícula p perteneciente al tejido y determinada por sus coordenadas (i_p, j_p) en éste, la trayectoria que ésta seguirá en el espacio viene determinada por la ecuación

$$\begin{aligned} \vec{y}_p''(t) &= F(t, \vec{y}_p(t), \vec{y}_p'(t)) = \\ &= \sum_{(i,j)\in E} \left(\frac{k_{ij}}{L_{ij}^2} \left(||\vec{y}_p(t) - \vec{y}_{ij}(t)|| - L_{ij} \right) \frac{\vec{y}_p(t) - \vec{y}_{ij}(t)}{||\vec{y}_p(t) - \vec{y}_{ij}(t)||} + \frac{d_{ij}}{L_{ij}^2} \left(\vec{y}_p'(t) - \vec{y}_{ij}'(t) \right) \right) \end{aligned} \quad (2.9)$$

donde E es el conjunto de las partículas adyacentes a p y que, por tanto, influyen en la fuerza ejercida sobre ella.

De esta forma, únicamente habremos de partir de las funciones ya declaradas para el código del caso particular del muelle fijo, adaptándolas. En este caso la función `g`, que es la condicionada por las fuerzas del sistema, mientras que `f` continuará devolviendo la velocidad generada por `g`. Así pues, analíticamente todo lo declarado está bien definido pero, a la hora de llevar esto a un entorno de programación, lo más complicado, a primera vista, parece ser el poder determinar con exactitud este conjunto E establecido de forma abstracta. Sin embargo, esto no tiene mayor dificultad que la de razonar de forma detenida la situación a la que nos enfrentamos:

Observemos que, dada una partícula p situada en las coordenadas (i, j) del tejido a tratar, las partículas adyacentes a ella vienen determinadas, únicamente, por las coordenadas coloreadas de verde del siguiente esquema

$$\begin{array}{ccccc} (i-1,j-1) & \text{——} & (i-1,j) & \text{——} & (i-1,j+1) \\ | & & | & & | \\ (i,j-1) & \text{————} & (i,j) & \text{————} & (i,j+1) \\ | & & | & & | \\ (i+1,j-1) & \text{——} & (i+1,j) & \text{——} & (i+1,j+1) \end{array}$$

Siendo así, bastará con hacer una simple comprobación de la situación de la partícula p, definida por sus coordenadas (i, j), para sumar, caso de que exista la partícula a tratar, la fuerza correspondiente que ésta ejerza sobre la partícula p tratada. Cómo determinar si existe una tal partícula, y por tanto un muelle que le aplique su fuerza, es muy simple dado que, a partir de sus coordenadas, sabemos en qué fila y qué columna se encuentra. De este modo, si

dicha partícula está situada, por ejemplo, en el contorno superior del tejido, no habrá ninguna partícula por encima de ella, de manera que en el cómputo sólo se cuantificarán las fuerzas ejercidas por las partículas laterales e inferior.

En definitiva, construyendo la función `g` de forma que pasemos como parámetro el sistema completo de partículas y, las coordenadas de la correspondiente a tratar, podremos analizar con unos condicionales qué fuerzas se aplican sobre ésta. Particularizando y aplicando la correspondiente fuerza de manera vectorial sobre cada componente, la función `g` quedaría declarada de la siguiente manera:

```
float g(float t, float * y, float * v, char var, int i, int j) {
        float funcion = 0.0;
        int indiceD = i + COLUMNAS; //indice para fila inferior
        int indiceU = i - COLUMNAS; //indice para fila superior
        float X, Y, Z;
        float norma;
        if (var == 'x') {
                if (j != (COLUMNAS-1)) {//particula de la derecha
                        X = (y[4 * i + 4 * j] - y[4 * i + 4 * (j + 1)]);
                        Y = (y[4 * i + 4 * j + 1] - y[4 * i + 4 * (j + 1) + 1]);
                        Z = (y[4 * i + 4 * j + 2] - y[4 * i + 4 * (j + 1) + 2]);
                        norma = (float)sqrt(X*X + Y*Y + Z*Z);
                        funcion = funcion - k / (L*L) * (norma - L) * X / norma - d / (L*
                            L) * (v[3 * i + 3 * j] - v[3 * i + 3 * (j + 1)]);
                }
                if (indiceD != numparticles){//particula de abajo
                        X = (y[4 * i + 4 * j] - y[4 * indiceD + 4 * j]);
                        Y = (y[4 * i + 4 * j + 1] - y[4 * indiceD + 4 * j + 1]);
                        Z = (y[4 * i + 4 * j + 2] - y[4 * indiceD + 4 * j + 2]);
                        norma = (float)sqrt(X*X + Y*Y + Z*Z);
                        funcion = funcion - k / (L*L) * (norma - L) * X / norma - d / (L*
                            L) * (v[3 * i + 3 * j] - v[3 * indiceD + 3 * j]);
                }
                if (j != 0) {//particula de la izquierda
                        X = (y[4 * i + 4 * j] - y[4 * i + 4 * (j - 1)]);
                        Y = (y[4 * i + 4 * j + 1] - y[4 * i + 4 * (j - 1) + 1]);
                        Z = (y[4 * i + 4 * j + 2] - y[4 * i + 4 * (j - 1) + 2]);
                        norma = (float)sqrt(X*X + Y*Y + Z*Z);
                        funcion = funcion - k / (L*L) * (norma - L) * X / norma - d / (L*
                            L) * (v[3 * i + 3 * j] - v[3 * i + 3 * (j - 1)]);
                }
                if (i != 0) {//particula arriba
                        X = (y[4 * i + 4 * j] - y[4 * indiceU + 4 * j]);
                        Y = (y[4 * i + 4 * j + 1] - y[4 * indiceU + 4 * j + 1]);
                        Z = (y[4 * i + 4 * j + 2] - y[4 * indiceU + 4 * j + 2]);
                        norma = (float)sqrt(X*X + Y*Y + Z*Z);
                        funcion = funcion - k / (L*L) * (norma - L) * X / norma - d / (L*
                            L) * (v[3 * i + 3 * j] - v[3 * indiceU + 3 * j]);
                }
        }
.
.
.
}
```

Cabe destacar que, en lo referido al acceso a memoria de las coordenadas, recordemos que cada partícula viene determinada por cuatro espacios de memoria correspondiente a las coordenadas y a la inversa de la masa de la partícula en cuestión. Así, cada vez que avancemos una partícula avanzamos cuatro espacios en memoria, por ello, será necesario multiplicar por cuatro las coordenadas que definan la partícula, y sumarle 0, 1 ó 2, según si la fuerza calculada va referida a la coordenada x, y o z, respectivamente. El caso presentado es el referido a la coordenada x, para el resto de coordenadas es análogo.

Otro detalle a destacar es cómo llevar esto a cabo en según qué método. En nuestro método Euler Explícito, los datos que pasamos a nuestras funciones `f` y `g` son directamente los del

sistema que estamos tratando en cada interación. Sin embargo, en el caso de los métodos Runge-Kutta, es necesario calcular distintas pendientes, y para ello distintos sistemas, para hacer el promedio final. Observemos que, para el cálculo de la tercera pendiente de Runge-Kutta de cuarto orden, su cálculo analítico viene establecido, en nuestro ejemplo (en la coordenada x), como

$$k3_{i,j,x} = f(t+\frac{h}{2}, \vec{x}(t) + k2_{i,j,x}\frac{h}{2}, \vec{x}'(t) + l2_{i,j,x}\frac{h}{2}, x) \quad (2.10)$$

$$l3_{i,j,x} = g(t+\frac{h}{2}, \vec{x}(t) + k2_{i,j,x}\frac{h}{2}, \vec{x}'(t) + l2_{i,j,x}\frac{h}{2}, x) \quad (2.11)$$

donde nuestros parámetros $\vec{x}$ y $\vec{x}'$ son vectoriales, y donde se efectuarán los cálculos como los descritos antes en el código. Por ello, será necesario pasar un vector auxiliar de posiciones y de velocidades cuyas componentes hayan sido todas desplazadas por igual, con el fin de que las fuerzas ejercidas por los muelles entre las partículas sean coherentes y referidas al mismo sistema, es decir, a la misma configuración de partículas para el tejido en el tiempo t.

De este modo, para cada iteración, bastará hacer el cálculo completo de todas las componentes de cada pendiente k_i por separado. Así, antes de calcular el valor correspondiente a cada pendiente, establecer la variación completa del vector posición y velocidad, con los parámetros ya calculados de la pendiente anterior. De forma que, en lugar de tener el esquema (2.10), éste se transforma en el esquema

$$k3_{i,j,x} = f(t+\frac{h}{2}, \overrightarrow{xAux}(t), \overrightarrow{xAux}'(t), x)$$

$$l3_{i,j,x} = g(t+\frac{h}{2}, \overrightarrow{xAux}(t), \overrightarrow{xAux}'(t), x)$$

donde $\overrightarrow{xAux}(t)$ y $\overrightarrow{xAux}(t)$ han sufrido la correspondiente variación posicional que establece el método para dicha pendiente. Realizando esto para cada una de las pendientes que entran en juego, nuestra función `RungeKutta4` vendrá declarada como

```
void RungeKutta4() {
        float *auxPos;
        float *auxVel;
        float auxiliarCloth[4 * numparticles];
        float auxiliarVel[4 * numparticles];

        float k1[3 * numparticles], k2[3 * numparticles], k3[3 * numparticles], k4[3 *
            numparticles];
        float l1[3 * numparticles], l2[3 * numparticles], l3[3 * numparticles], l4[3 *
            numparticles];
        char var[] = { 'x', 'y', 'z' };

        int posi;
        int posj;

        //Para k1
        for (int i = 0; i < FILAS; i++) {
                for (int j = 0; j < COLUMNAS; j++) {
                        for (int h = 0; h < 3; h++) {
                                posi = i * COLUMNAS;
                                posj = j;

                                k1[3 * posi + 3 * posj + h] = f(simulationtime,
                                    currentcloth[4 * posi + 4 * posj + h], currentvel[3 *
                                     posi + 3 * posj + h],
                                        var[h]);
                                l1[3 * posi + 3 * posj + h] = g(simulationtime,
                                    currentcloth, currentvel, var[h], posi, posj);
                        }
                }
        }
```

```
        //Para k2
        copiaVector(currentcloth, auxiliarCloth, 4);
        system("pause");*/
        sumaVector(auxiliarCloth, 4, k1, TIME_STEP / 2.0f);
        copiaVector(currentvel, auxiliarVel, 3);
        sumaVector(auxiliarVel, 3, l1, TIME_STEP / 2.0f);
        for (int i = 0; i < FILAS; i++) {
                for (int j = 0; j < COLUMNAS; j++) {
                        for (int h = 0; h < 3; h++) {
                                posi = i * COLUMNAS;
                                posj = j;

                                k2[3 * posi + 3 * posj + h] = f(simulationtime +
                                    TIME_STEP / 2.0f, auxiliarCloth[4 * posi + 4 * posj +
                                     h],
                                        auxiliarVel[3 * posi + 3 * posj + h], var[h]);
                                l2[3 * posi + 3 * posj + h] = g(simulationtime,
                                    auxiliarCloth, auxiliarVel, var[h], posi, posj);
                        }
                }
        }

        //Para k3
        copiaVector(currentcloth, auxiliarCloth, 4);
        sumaVector(auxiliarCloth, 4, k2, TIME_STEP / 2.0f);
        copiaVector(currentvel, auxiliarVel, 3);
        sumaVector(auxiliarVel, 3, l2, TIME_STEP / 2.0f);
        for (int i = 0; i < FILAS; i++) {
                for (int j = 0; j < COLUMNAS; j++) {
                        for (int h = 0; h < 3; h++) {
                                posi = i * COLUMNAS;
                                posj = j;

                                k3[3 * posi + 3 * posj + h] = f(simulationtime +
                                    TIME_STEP / 2.0f, auxiliarCloth[4 * posi + 4 * posj +
                                     h],
                                        auxiliarVel[3 * posi + 3 * posj + h], var[h]);
                                l3[3 * posi + 3 * posj + h] = g(simulationtime,
                                    auxiliarCloth, auxiliarVel, var[h], posi, posj);
                        }
                }
        }
        .
        .
        .
}
```

donde la función `copiaVector` se encarga de copiar un vector indicado en el primer parámetro, a otro, indicado en el segundo, que tiene un número entero de coordenadas referidas a cada partícula. En el caso de las posiciones es cuatro por llevar la inversa de la masa determinada, y en el caso de la velocidad es tres, por referirse únicamente a las coordenadas (x, y, z). De igual modo se define la función `sumaVector`, que suma a un vector, indicado en el primer parámetro, otro multiplicado por un coeficiente, precisado en el último parámetro.

Finalmente, tomando como parámetros

```
/*Constante de MUELLE*/
float m = 1.0;
float L = 4.0/(COLUMNAS-1);
float k = 1000.0;
float d = 1.00;
float gravity = -9.8;
float x_0 = -1.0;
float y_0 = 3.0;
float v_0 = 0.0;
float z_0 = 0.0;
```

estableciendo como posiciones iniciales: una distribución simétrica al centro según, las columnas establecidas, en el eje x; posición y idéntica para todas las partículas, y posición en z desde el origen hacia el lado negativo.

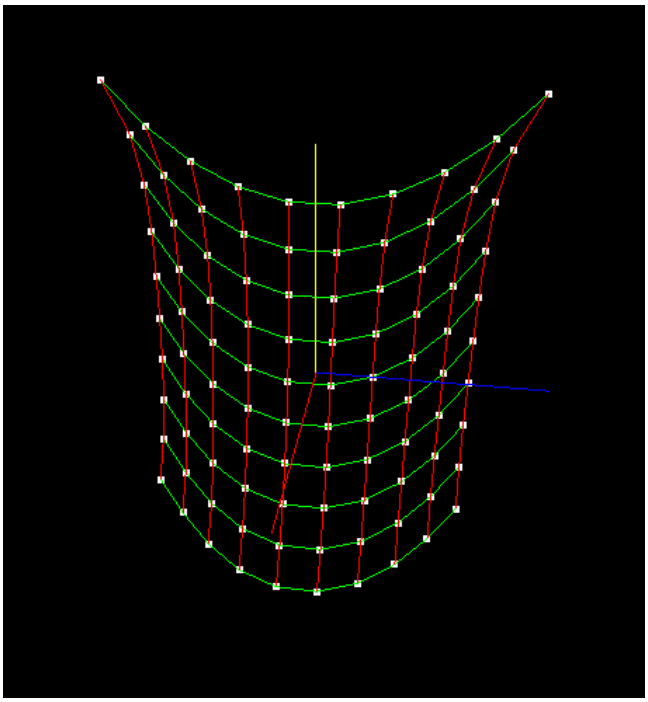


Figura 2.16: Sistema de partículas conectadas entre sí mediante muelles, suspendido de sus dos extremos superiores.

Aplicando además la fuerza de la gravedad en la función `g`, obtenemos una simulación que asemeja perfectamente el movimiento de un tejido suspendido sobre dos de sus extremos, el cuál podemos contemplar en la **Figura 2.16**.

2.7. Aplicaciones al código fuente

Llegados a este punto podemos comprobar cómo, a partir de métodos numéricos que nos permitían en el Capítulo 1 aproximar ciertas soluciones de Ecuaciones Diferenciales Ordinarias, paso a paso, con estas aproximaciones, hemos sido capaces crear objetos y formas en 3D que se rijan por distintas leyes físicas. Desde los ejemplos como el Shampine Flame, mera aproximación a un proceso físico como el encendido de una llama, hemos pasado a resolver situaciones en cada vez más dimensiones del espacio, llegando a realizar las impresionantes simulaciones de los atractores, o el sistema en 3D de partículas enlazadas con muelles. Aproximándonos así, cada vez más, a nuestro objetivo de Realidad Virtual.

Habiendo procedido de esta manera, y con la seguridad que nos proporcionan las comprobaciones y ejemplos realizados, no cabe más que utilizar todo ello en un último paso final: el paso a la "Realidad Virtual".

Lo primero que hicimos en el capítulo, una vez habíamos comprendido completamente el código, fue simplificarlo y adecuarlo mejor a nuestra forma de trabajo. A partir de ahí, desarrollamos los ejemplos que nos condujerons, finalmente, a la elaboración de un código que computara la ley de Hooke para un sistema de partículas fijo. Trabajando ahora con estos dos códigos como

base, implementaremos todos los métodos desarrollados en los diferentes ejemplos, junto con el Verlet, a fin de comprobar qué cambios se producen con respecto al código original con el que empezamos a trabajar, y contrastaremos los resultados.

Comencemos en primer lugar por añadir al código los métodos de `EulerExplicito`, `RungeKutta2` y `RungeKutta4`.

Puesto que en el código original interviene, fundamentalmente, tres fuerzas: gravedad, rozamiento y fuerza del viento; será necesario incluir éstas en la función `g` que ya teníamos definida. Dado que posteriormente puede interesarnos aplicar o no la Ley de Hooke, dejaremos esta declarada tal cuál la teníamos y añadiremos, tras lo referente a esta fuerza, las líneas de código (divididas entre coordenadas) relacionadas con las fuerzas que queremos añadir. De esta manera bastará con sumar sus contribuciones a las ya declaradas siendo 0.0 si no hay aportación por Hooke, o la correspondiente en caso de haberla.

Así, añadimos las siguientes líneas de código a nuestra función `g`

```
//Fuerza Rozamiento
        float coef = -d;
        float dampforce = 0.0;
        if (!Hooke) {
                dampforce = v[3 * i + 3 * j] * normal[3 * i + 3 * j]
                        + v[3 * i + 3 * j + 1] * normal[3 * i + 3 * j + 1]
                        + v[3 * i + 3 * j + 2] * normal[3 * i + 3 * j + 2];
        }
        //Fuerza viento
        float windforce;
        //Determina el vector fuerza del viento
        windvector[0] = winddirection[0] * windspeed;
        windvector[1] = winddirection[1] * windspeed;
        windvector[2] = winddirection[2] * windspeed;

        windforce = windvector[0] * normal[3 * i + 3 * j]
                                + windvector[1] * normal[3 * i + 3 * j + 1]
                                + windvector[2] * normal[3 * i + 3 * j + 2];

        if (var == 'x') {
                funcion = funcion + coef * dampforce * normal[3 * i + 3 * j]
                                + windforce * normal[3 * i + 3 * j];
        }
        else if (var == 'y') {
                funcion = funcion + gravity + coef * dampforce * normal[3 * i + 3 * j +
                    1]
                                + windforce * normal[3 * i + 3 * j + 1];
        }
        else if (var == 'z') {
                funcion = funcion + coef * dampforce * normal[3 * i + 3 * j + 2]
                                + windforce * normal[3 * i + 3 * j + 2];
        }
```

destacando que la gravedad únicamente se aplica a la coordenada y. Como podemos observar será necesario indicar, con un pequeño condicional, si se aplica la Ley de Hooke o no, ya que dentro del cálculo de esta ley ya se aplica el correspondiente rozamiento de la partícula, de modo que, si lo utilizamos de nuevo, lo estaremos incluyendo dos veces. Sin embargo, si no aplicamos Hooke, será necesario calcularla y, para ello, utilizaremos el mismo coeficiente que para dicha ley.

Señalado esto hay otra cuestión que es necesario resolver. Cuando en el código original la esfera entra en contacto con el tejido, éste no la atraviesa puesto que la función `satisfaycontraints` se encarga de establecer unas restricciones para expulsar la partícula fuera de la esfera en caso de que ésta entre en ella. Sin embargo, cuando nosotros sustituímos esta función por nuestra

ley de Hooke, no hay restricciones que rijan la interacción entre la esfera y la tela. Para resolver esto será suficiente con añadir la función `restriccionEsfera`

```
void restriccionEsfera() {
        float deltaX;
        float deltaY;
        float deltaZ;
        float distancia;
        for (int i = 0; i < numparticles; i++) {
                deltaX = currentcloth[4 * i] - sphere[0];
                deltaY = currentcloth[4 * i + 1] - sphere[1];
                deltaZ = currentcloth[4 * i + 2] - sphere[2];

                distancia = deltaX * deltaX + deltaY * deltaY + deltaZ * deltaZ;
                if (distancia < sphere[3] * sphere[3]) {
                        currentcloth[4 * i] += deltaX * (sphere[3] - distancia) / sphere
                            [3];
                        currentcloth[4 * i + 1] += deltaY * (sphere[3] - distancia) /
                            sphere[3];
                        currentcloth[4 * i + 2] += deltaZ * (sphere[3] - distancia) /
                            sphere[3];
                }
        }
}
```

que será la encargada de aplicar las restricciones mencionadas a las partículas. Llamaremos a esta función al final de cada método numérico para que se compruebe tras cada iteración, una vez calculadas las nuevas posiciones. Así, antes de visualizar nada por pantalla, nos aseguraremos de que los resultados observados sean coherentes, de lo contrario se actualizarán las posiciones debidamente, expulsando las partículas que se encuentren en el interior de la esfera hasta su controno.

2.8. Conclusiones y contraste de Resultados

Para concluir, contrastaremos el programa principal con las mejoras que hemos realizado sobre éste. Si bien, además de RK4, también implementamos RK2 y Euler Explícito, lo que buscamos es tener un método eficiente y una dinamización de la visualización ágil. Por ello, las comparaciones las centraremos únicamente en RK4 y Verlet puesto que, como sabemos y queda debidamente justificado en el capítulo, estos son mucho más eficientes que los anteriores, y no merece la pena considerar el resto de casos.

Así pues, para que los resultados que obtengamos puedan ser comparables, fijaremos un intervalo de tiempo y variaremos el paso a fin de poder tener controlada la situación que tratemos y ser capaces, así, de contrastar de manera fiable las conclusiones. Además, simplificaremos la situación: puesto que Verlet es un método el cual no hace uso de las velocidades, suprimiremos éstas tomando la constante de rozamiento como 0.0 de forma que la única contribución que afecta a la velocidad no se tenga en cuenta. Por otro lado, para evitar que agentes externos varíen los resultados, sacaremos a nuestra esfera de escena comparando, así, dos sistema totalmente idénticos: uno ejecutado con RK4 y, el otro, con Verlet.

Tomando, pues, $T = 2.0$, para realizar las comparaciones procederemos de la siguiente manera:

Para distintos valores del paso de tiempo h, contrastaremos la posición final del paño para ambos métodos obteniendo, en norma, el máximo de las diferencias entre ambas. De esta manera tomaremos dos vectores $yRK4$ e $yVerlet$, de dimensiones $4 * numparticulas$, tales que

$yRK4^t_{p,i}$ corresponde a la coordenada i de la partícula p para RK4 en el tiempo t

y, análogamente

$$yVerlet^t_{p,i} \quad \text{para Verlet}$$

donde $i = 0, 1, 2$ corresponde a la coordenada x, y, z respectivamente. Guardando en un archivo externo que podamos ejecutar con MATLAB los datos de $yRK4^T$ e $yVerlet^T$, podemos realizar la norma del máximo para la diferencia, tal que

$$||yRK4^T - yVerlet^T|| = max_p||yRK4^T_p - yVerlet^T_p|| = max_p\Big(max_i|yRK4^T_{p,i} - yVerlet^T_{p,i}|\Big)$$

Trasladando dicho procedimiento analítico a código en MATLAB, podemos hacer una comparativa de la diferencia de la posición final del paño obtenida con cada método.

Para ello, necesitaremos establecer los pasos adecuados para los dos métodos en cada comparación puesto que, siendo Verlet de orden dos, $error \approx O(h^2)$, y Runge-Kutta cuatro de cuarto, $error \approx O(h^4)$, para una misma situación en T hemos de buscar un h_V y un h_{RK} tales que $O(h_V^2) = O(h_{RK}^4)$. Para conseguir esto, jugaremos con las particiones del intervalo consideradas en cada caso:

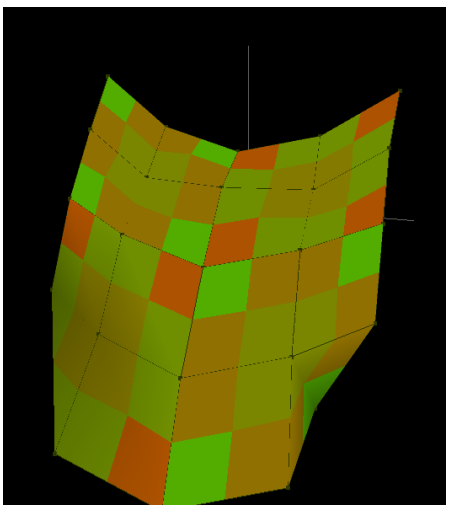
Nuestro paso h se obtiene del cociente entre $\frac{T}{N}$ de forma que $h_V = \frac{T}{N_V}$ y $h_{RK} = \frac{T}{N_{RK}}$ y, puesto que lo que buscamos es $O(h_V^2) = O(h_{RK}^4)$, entonces se tendrá que $\frac{T^2}{N_V^2} = \frac{T^4}{N_{RK}^4}$. Luego, podemos determinar que, dado una partición N_{RK} para el método de RK4, se obtendrá un error del mismo orden para Verlet utilizando una partición del intervalo del tamaño $N_V = \frac{N_{RK}^2}{T}$.

En estas condiciones ya podemos realizar nuestro estudio adecuadamente.

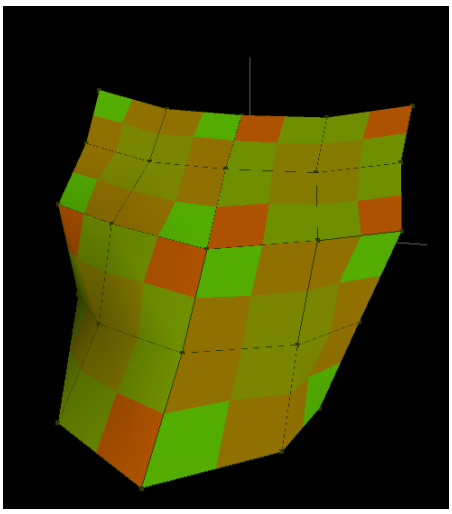
Partiendo del programa al que hemos añadido la ley de Hooke, sustituyendo la función `satisfayconstraints`, vamos a ver para el tiempo establecido cuáles son las diferencias al acabar la simulación entre los métodos. El objetivo de esta comparación es poder observar si, efectivamente, el método de Runge-Kutta que hemos añadido presenta una mejora en éste o, por el contrario, sigue siendo más efectivo un método simpléctico como el Verlet.

Los resultados que hemos obtenido son los que quedan reflejados en la siguiente tabla:

N_{RK}	N_V	h_{RK}	h_V	diferencia
50	1250	0.04	0.0016	0.6462
80	3200	0.025	$6.25 \cdot 10^{-4}$	0.2104
100	5000	0.02	0.0004	0.0836



Simulación con Ley de Hooke para $N = 50$ con RK4

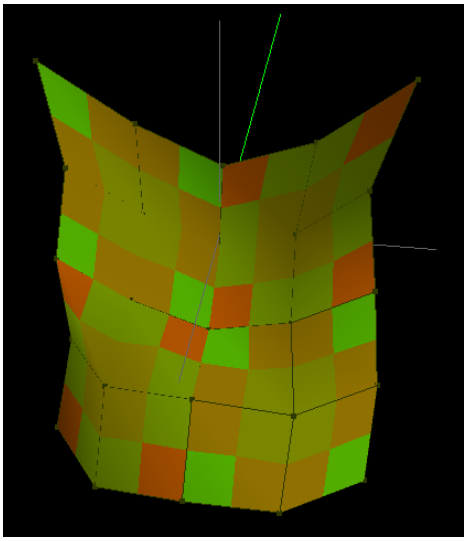


Simulación con Ley de Hooke para $N = 1250$ con Verlet

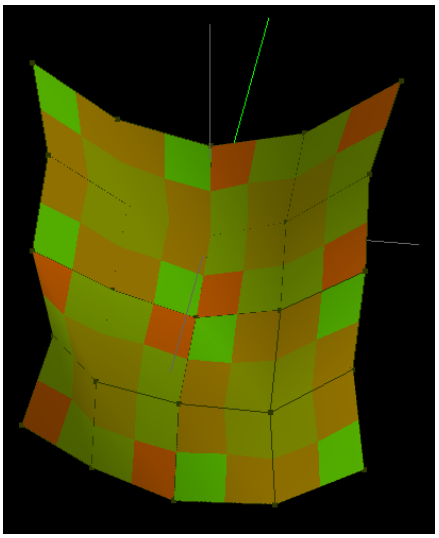
Realizada esta primera comprobación podemos observar que las diferencias entre el método de RK4 y Verlet no son, ni mucho menos, desorbitadas para un tiempo fijo establecido. Como podemos observar en la imagen, la situación obtenida al final del cálculo es realmente parecida entre ambos resultados. A pesar de ello, no nos detendremos aquí, y puesto que hay más elementos que intervienen en nuestro ejemplo base, veamos qué ocurre cuando añadimos más propiedades al sistema.

Analizando ahora otra de las fuerzas que intervienen en el código original: la fuerza del viento. Ésta no depende de la velocidad del sistema, por lo que nos encontramos en las condiciones anteriores establecidas. De esta forma, tomando una velocidad de viento de magnitud 10, que permite, en este intervalo de tiempo, observar un cambio drástico en la simulación, obtenemos los datos reflejados en la siguiente tabla:

N_{RK}	N_V	h_{RK}	h_V	diferencia
50	1250	0.04	0.0016	0.3461
80	3200	0.025	$6.25 \cdot 10^{-4}$	0.1449
100	5000	0.02	0.0004	0.0906



Simulación con Ley de Hooke y viento para $N = 50$ con RK4



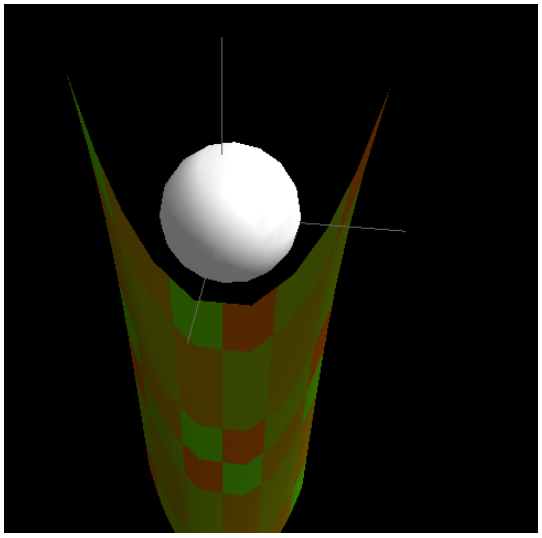
Simulación con Ley de Hooke y viento para $N = 1250$ con Verlet

Como podemos observar también aquí, la diferencia que obtenemos entre los resultados finales de cada métodos son, apenas, apreciables en nuestra simulación.

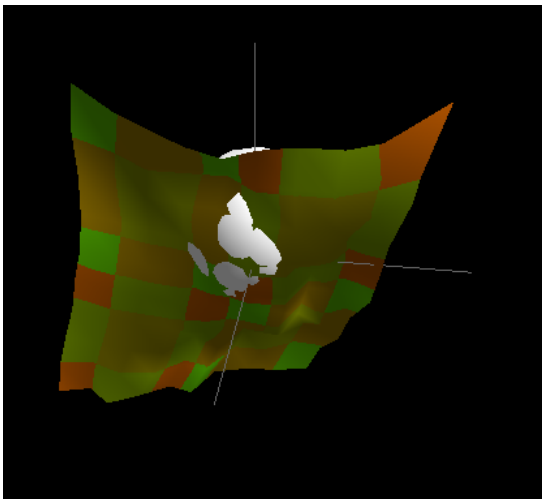
Finalmente, tras todas las pruebas efectuadas y las comparaciones establecidas podemos concluir que, llevando una simulación renderizada únicamente con procesos físicos, el método RK4 que hemos introducido, al ser de orden superior, es mucho más eficiente que el Verlet. Como podemos comprobar en las diferentes tablas, para pasos mucho más grandes que los requeridos para Verlet, nuestro método RK4 nos da situaciones prácticamente iguales que las obtenidas con Verlet. De este modo queda evidenciado que, en tiempo finito, todos los resultados que hemos llevado a cabo durante el capítulo, aplicados al código original, han presentado una mejora en él en tiempo finito, que era el objetivo de todo nuestro estudio. Sin embargo, no podemos obviar el estudio realizado en el capítulo 1. Aunque en el código hemos introducido grandes mejoras como la ley de Hooke, no podemos olvidar que RK4 pierde energía en el sistema a lo largo de su computación, mientras que Verlet, por ser simpléctico, la conserva. De modo que, aunque las aproximaciones con RK4 en tiempo finito son mejores, para tiempos prolongados habremos de

utilizar un paso mucho más pequeño y hacer uso del método de Verlet, que nos garantizará la consistencia del problema.

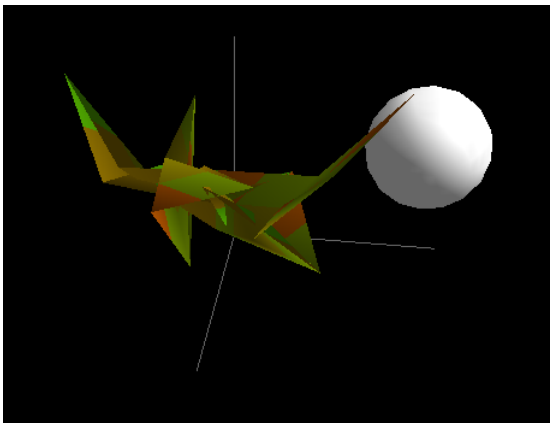
Visto todo esto, podemos observar algunos de los ejemplos de las inestabilidades que nos hemos ido encontrando a lo largo de las simulaciones, como son los siguientes

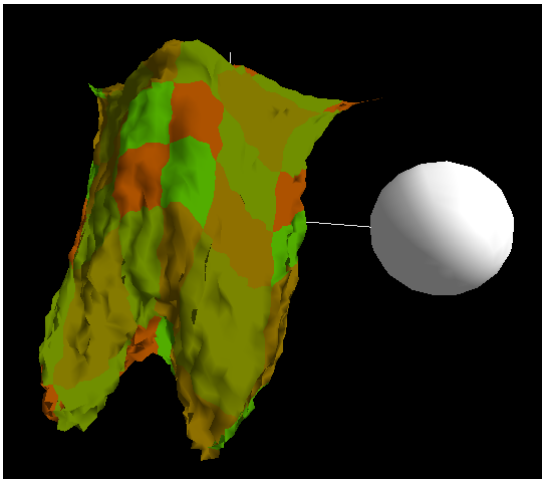


Inestabilidad producida al establecer la longitud de reposo del muelle más larga que la distancia originada por las posiciones iniciales.



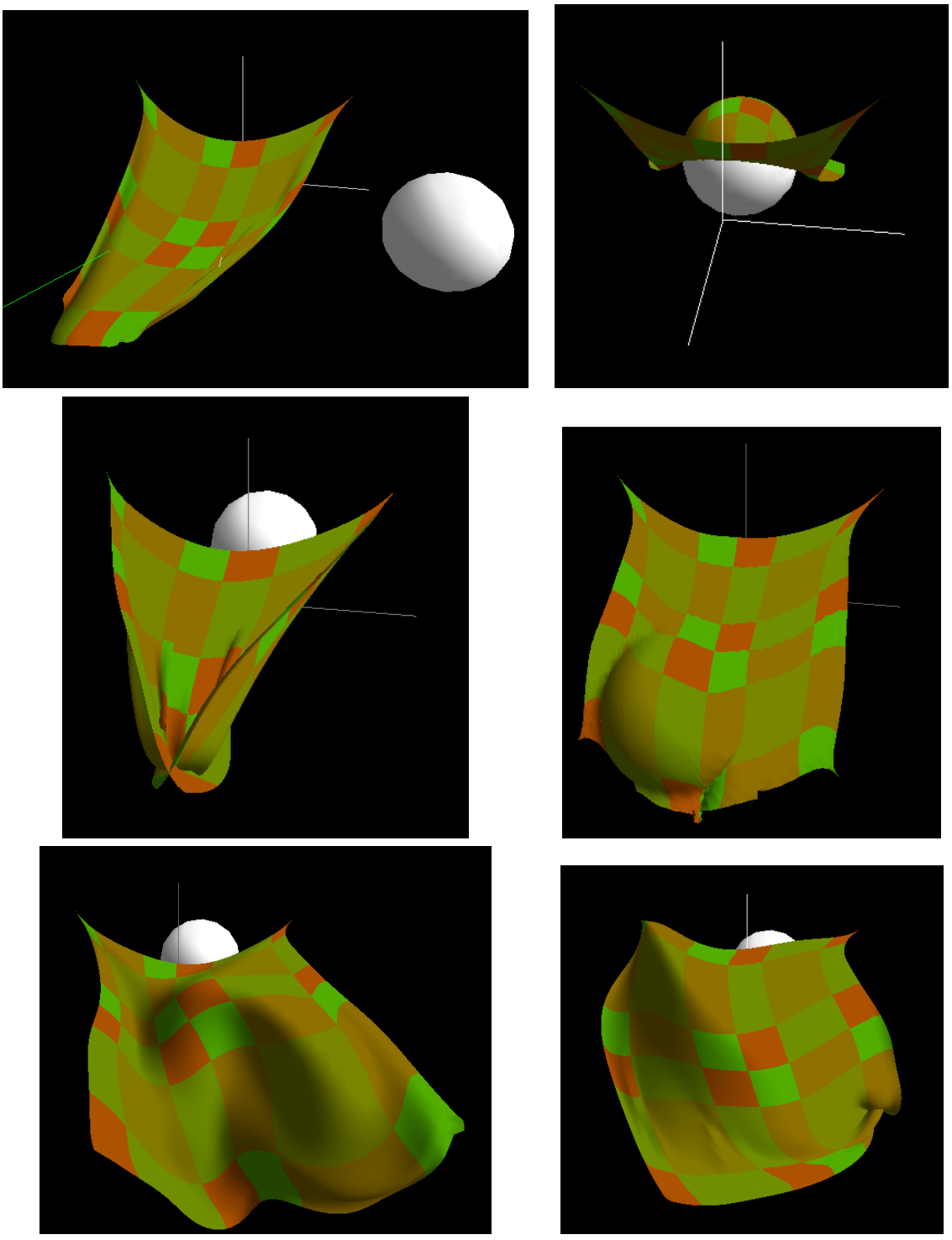
Inestabilidad producida al establecer un número de partículas demasiado pequeño, dando lugar así grandes a huecos entre ellas que atraviesa la esfera.





Inestabilidades producidas al establecer un paso de tiempo demasiado grande en relación al número de partículas que conforman el sistema. A la izquierda se observa el sistema estallando y, a la derecha, justo antes de estallar.

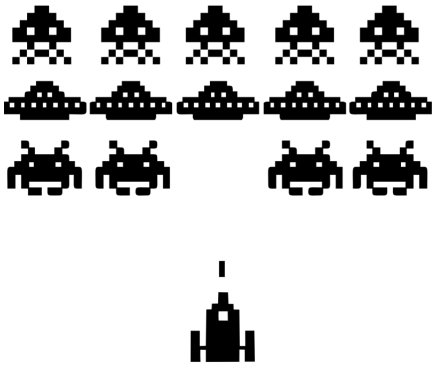
Para terminar, con motivo de finalización del capítulo y como muestra de los frutos obtenidos, mostramos algunos ejemplos de la simulación con distintos distintos valores de paso de tiempo, en distintos casos. Dichos ejemplos están llevados a cabo para un número fijo de partículas, para conformar un paño lo suficientemente semejante a la realidad, y que nos ofreciera una renderización ágil. Para ello hemos empleado 2500 partículas, además de utilizar, como método de cálculo, el método de RK4 que hemos desarrollado y, el cual, nos permite hacer, a pasos más grandes, una simulación más ágil y exacta que el método de Verlet del código original. Así, obtenemos los siguientes resultados:



Del mismo modo, y aprovechando la versatilidad del código que hemos conseguido realizar, en el **Anexo 1** podemos encontrar una serie de dinamizaciones curiosas obtenidas al fijar distintos puntos que, dada la finalidad de nuestro trabajo, y el juego que ofrece, hemos considerado interesante añadir.

De esta forma terminamos un elaborado programa que, como el original, nos permite observar la interacción de un paño con una esfera mediante la iteración del método numérico de Runge-Kutta de cuarto orden. Obteniendo así un paño dinámico, con movimientos suaves y sutiles, gracias a la simulación de éste con un elevado número de partículas, conectadas por muelles, los cuales otorgan la elasticidad apropiada del tejido que podemos observar en nuestro programa.

Lo que hemos llevado a cabo en este trabajo no es más que una muestra de los grandes avances que está consiguiendo la tecnología en nuestros días. Éstas, y otras técnicas mucho más elaboradas, son las que podemos encontrar hoy en día en un mundo cada vez más conocido y popular: el mundo virtual. Todos recordamos aquél juego con el que comenzamos a jugar de pequeños, la primera maquinita que ya nos alucinaba al ver elementos en movimiento en un dispositivo, el famoso juego de los "marcianitos". Todo ello ha ido evolucionando de una manera asombrosa hasta nuestros días, llegando a reproducir elementos de la vida cotidiana, como el paño de nuestro ejemplo, en un universo virtual.



Esto ha dado lugar a implementar estos avances en mundos como el del cine o, incluso, en los videojuegos, llevando la animación a puntos inimaginables: hemos pasado de tratar con elementos como los de la imagen de la izquierda de los marcianitos, a llegar al punto de la imagen de la derecha, el videojuego "Assassin's Creed" de última generación, en el que tanto los entornos como los propios personajes tienen un nivel de detalle espectacular.

Bibliografía

[1] Guía glut - windows microsoft. https://msdn.microsoft.com/es-es/library/windows/desktop.

[2] Manual geogebra. https://www.geogebra.org/manual/es/Manual.

[3] Manual opengl-glut. https://www.opengl.org/documentation/specs/glut/spec3/node1.html.

[4] V.I. Arnold. *Mecánica Clásica. Métodos matemáticos*. PARANINFO S.A. Madrid, 1983.

[5] Santiago Boari. Simulaciones numéricas de atractores extraños. Dinámica No Lineal. Departamento de Física, Facultad de Ciencias Naturales y Exactas, Universidad de Buenos Aires, 2011.

[6] R.L. Burden and J.D. Faires. *Numerical Analysis*. Cengage Learning, 9 edition, 2010.

[7] J.C. Butcher. A history of runge-kutta methods. *Applied Numerical Mathematics*, 20(3):247 – 260, 1996.

[8] Ward Cheney and David Kincaid. *Numerical Mathematics and Computing*. International Thomson Publishing, 6th edition, 1998.

[9] Shalini Govil-Pai. *Principles of Computer Graphics. Theow and Practice Using OpenGL and Maya*. Springer, Berlin, 2004.

[10] D. Griffiths and D.J. Higham. *Numerical Methods for Ordinary Differential Equations: Initial Value Problems*. Springer Undergraduate Mathematics Series. Springer London, 2010.

[11] E. Hairer, S.P. Norsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer Series in Computational Mathematics. Springer Berlin Heidelberg, 2008.

[12] Michael Hauth and Olaf Etzmuss. A high performance solver for the animation of deformable objects using advanced numerical methods. *Computer Graphics Forum*, 20(3):319–328, 9 2001.

[13] M.H. Holmes. *Introduction to Numerical Methods in Differential Equations*. Texts in Applied Mathematics. Springer New York, 2006.

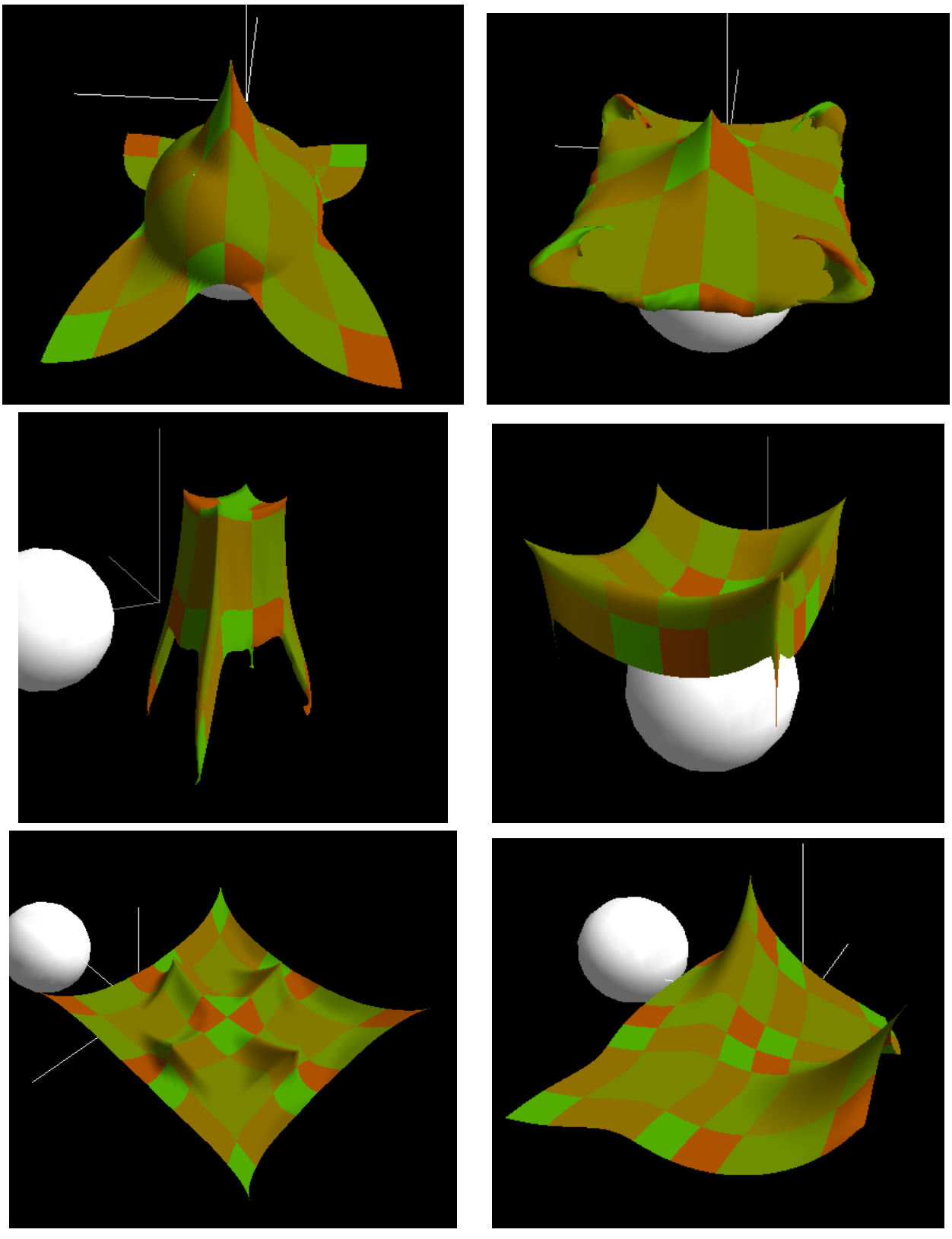
[14] Joe Pitt-Francis and Jonathan Whiteley. *Guide to Scientific Computing in C++*. Springer, 2012.

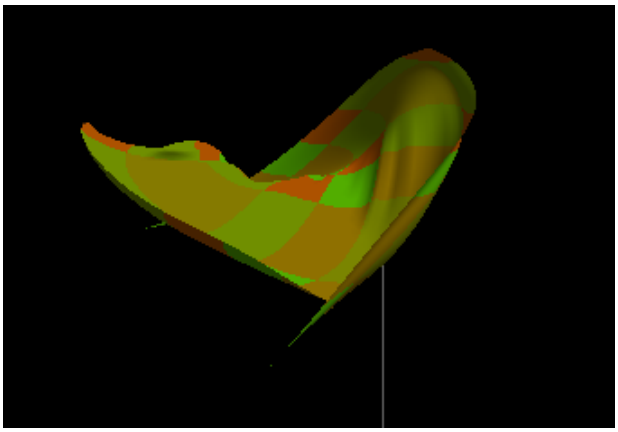
[15] Xavier Provot. Deformation constraints in a mass-spring model to describe rigid cloth behavior. In *In Graphics Interface*, pages 147–154, 1996.

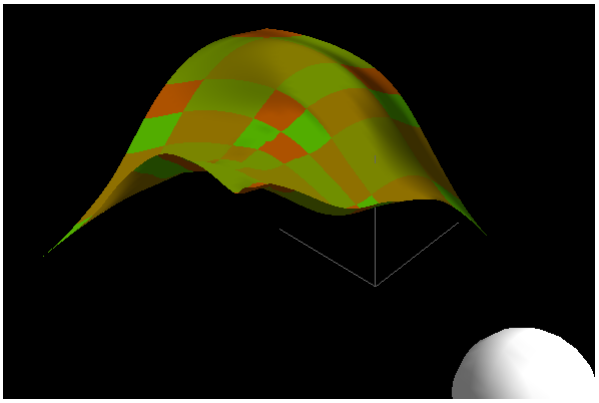
[16] A. Quarteroni, F. Saleri, and P. Gervasio. *Scientific Computing with MATLAB and Octave*. Texts in Computational Science and Engineering. Springer Berlin Heidelberg, 3 edition, 2010.

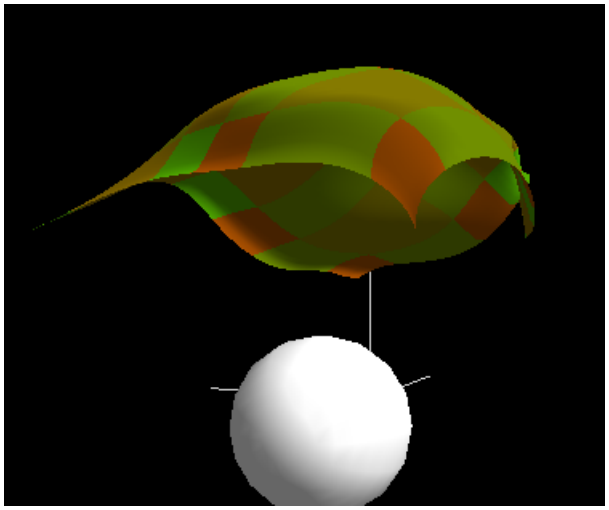
[17] H.D. Young, R.A. Freedman, F.W. Sears, A.L. Ford, J.E. Brito, M.W. Zemansky, and R.G. Leal. *Física universitaria con física moderna 12ED. Volumen II.* Física universitaria con física moderna. Pearson Educación, 12 edition, 2009.

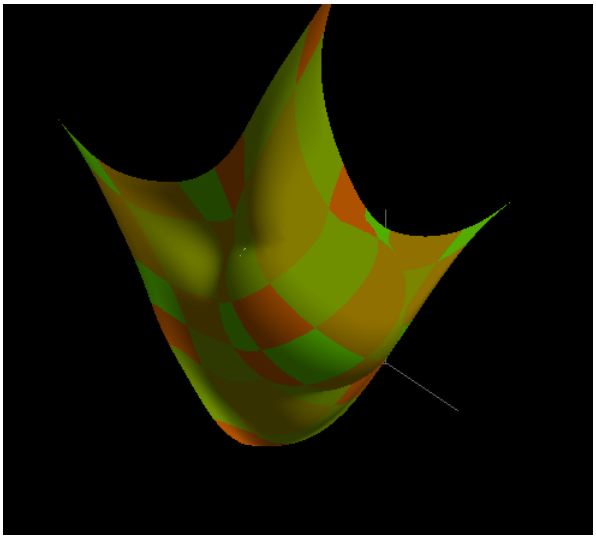
Anexo 1. Ejemplos de Dinamizaciones Destacables

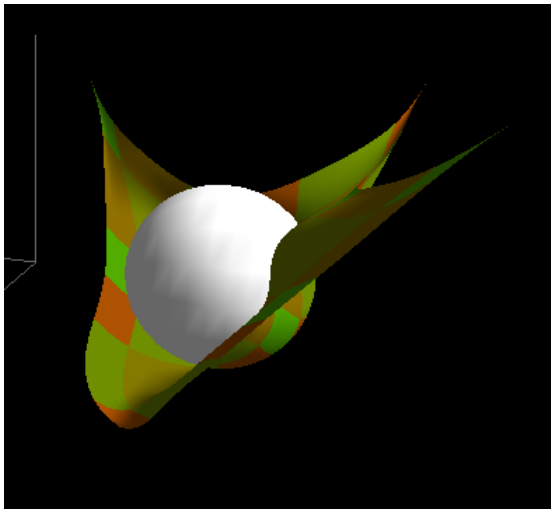


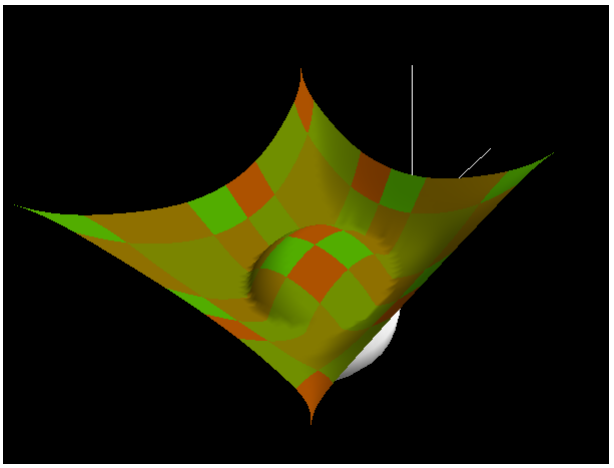


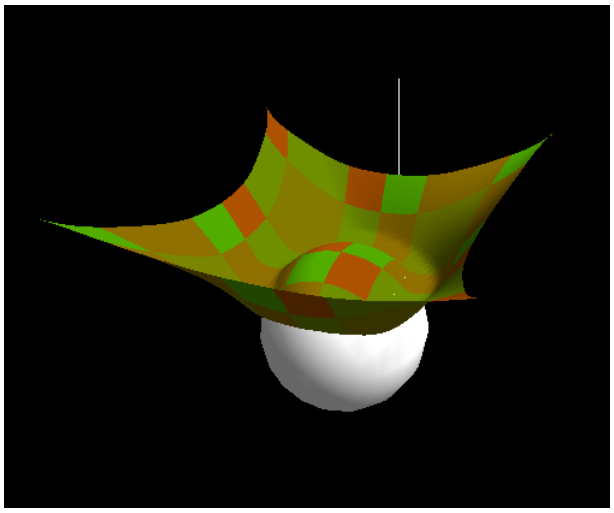


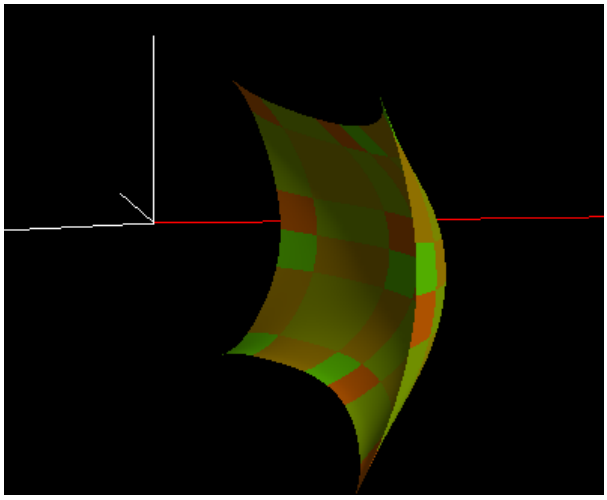


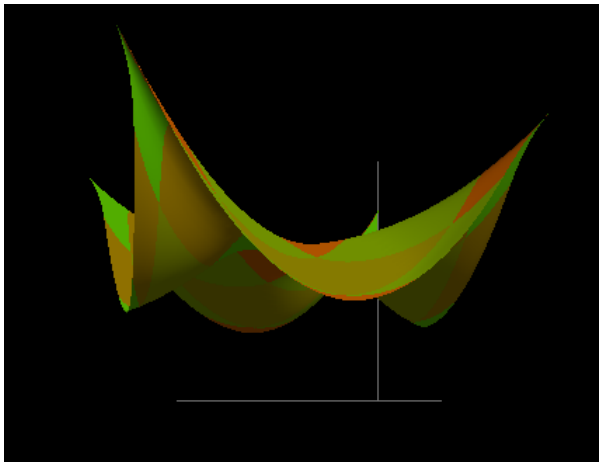


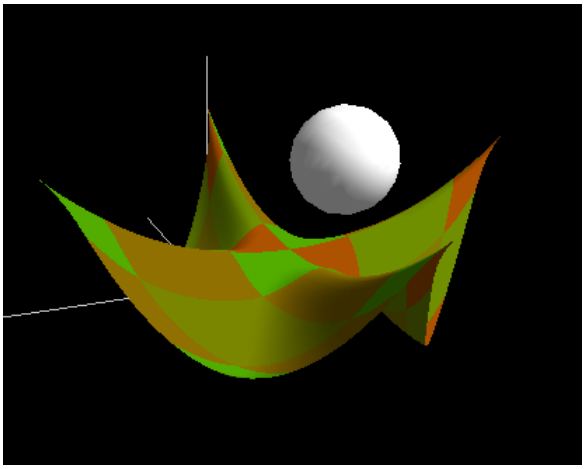


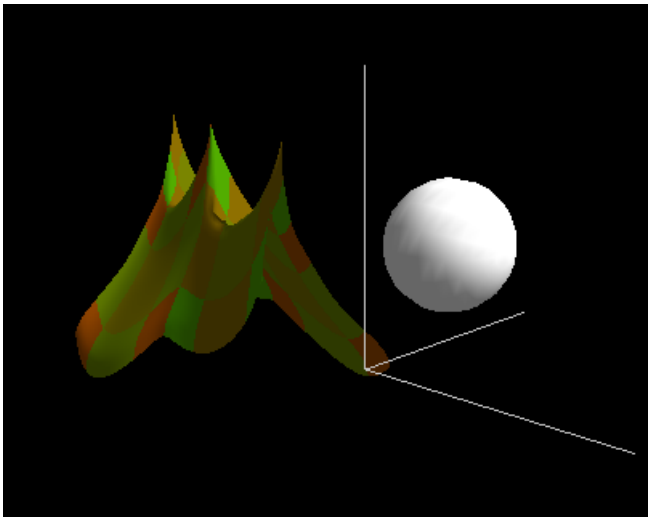


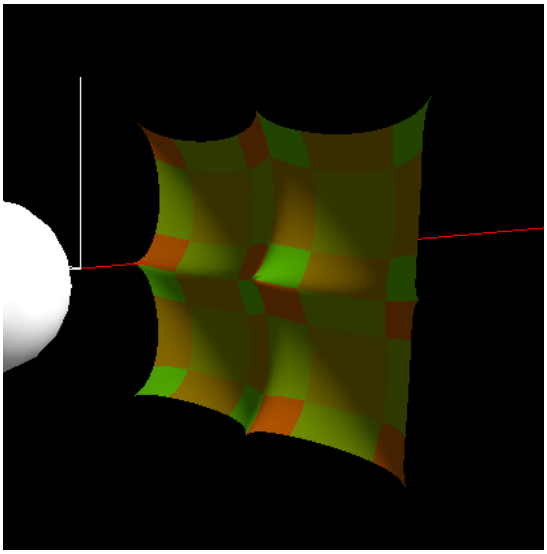


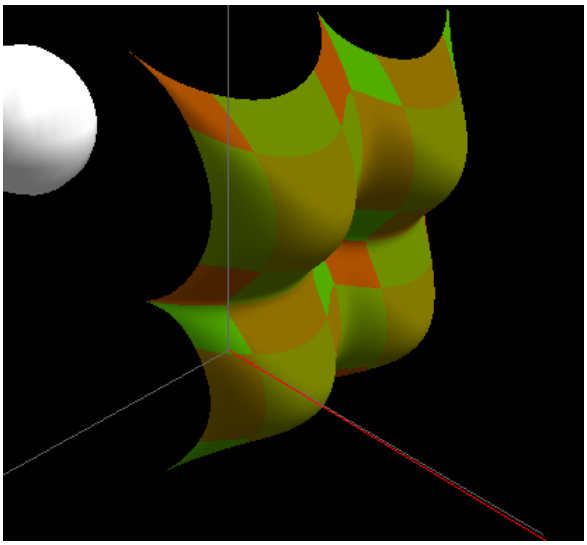


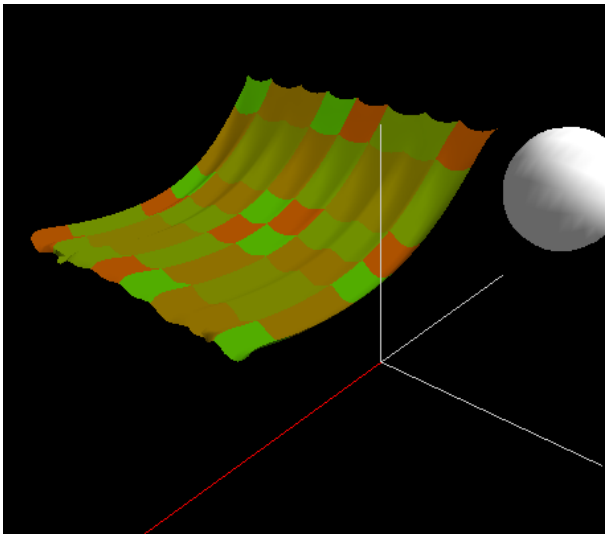


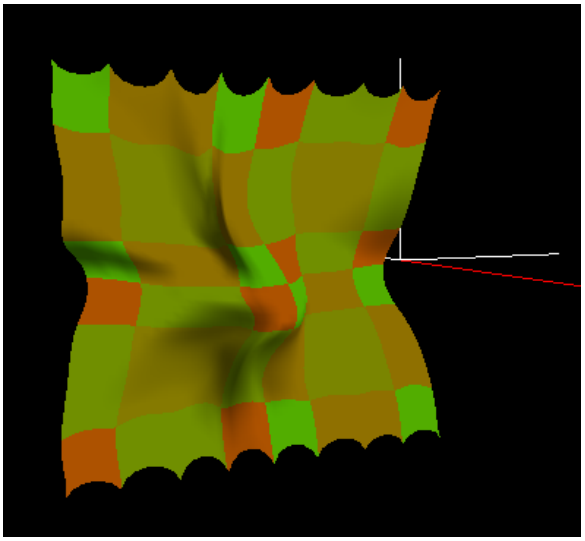


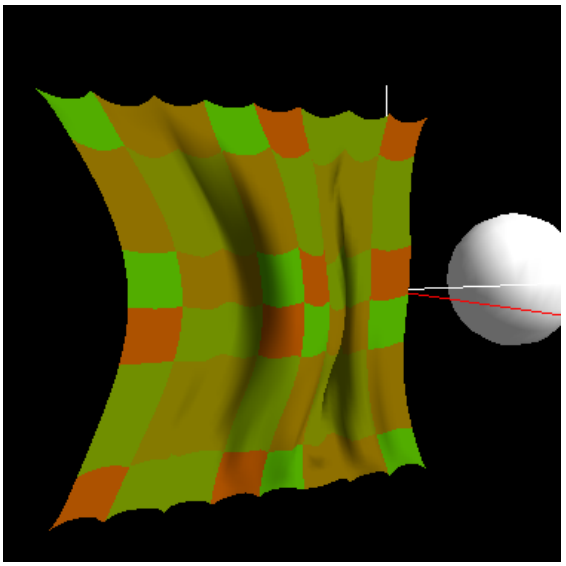












Anexo 2. Funciones Específicas de GLUT

- Función encargada de dibujar elementos en la pantalla:

```
void glBegin(GLenum mode);
```

Esta función delimita los vértices que definen un dato primitivo. Acepta un único argumento que especifica de cuál de los 10 modos se va a interpretar. A continuación presentamos algunos de ellos, los que forman parte de nuestros programas:

- ``mode = GL_TRIANGLE_STRIP'' .- Dibuja un grupo conectado de triángulos. Un triángulo esta definido por cada vértice presentado tras los primeros dos vértices. Para n impar: n, n+1 y n+2 define el triángulo n. Para n par: n+1, n, n+2 definen el triángulo n. En total se dibujarán N-2 triángulos.
- ``mode = GL_LINES'' .- Toma cada par de vértices como un segmento de línea independiente.
- ``mode = GL_POINTS'' .- Trata cada vértice como un punto singular. El vértice n define el punto n. Se dibujan N puntos.

Entre las delimitaciones ``glBegin'' y ``glEnd'' se sitúan los elementos entre los que queremos aplicar el modo indicado.

- Pasar coordenadas de la normal a cordenadas gl.

```
void glNormal3fv(const GLfloat *v );
```

Esta función asigna las coordenadas de un puntero de 3 elementos, referido a la normal indicada, a formato OpenGL para poder realizar la renderización.

- Pasar coordenadas de la puntos a cordenadas gl.

```
void glTexCoord2f(GLfloat x,GLfloat y);
```

Esta función establece las coordenadas de la textura que forma parte de los datos asociados con los vértices del polígono. En particular, especifica la textura en 2 dimensiones (x,y) de los parámetros pasados.

- Especificar el punto a dibujar

```
void glVertex3f(GLfloat x, GLfloat y, GLfloat z);
```

Esta función especifica el punto, línea y/o vértices del polígono que se va a dibujar. Las coordenadas del color actual, de la normal y de la textura se asocian al vértice indicado cuando llamamos a la función.

- Inicialización de la matriz identidad:

```
void glLoadIdentity(void);
```

Esta función reemplaza la matriz modelmatrix por la matriz identidad para establecer una nueva secuencia. Semánticamente equivale a llamar a la función `glLoadMatrix` pasando

como dato la matriz ([1,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]), sin embargo, en algunos casos, más eficiente.

- Generación de matriz visual:

```
void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble
    centerx, GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble
     upy, GLdouble upz);
```

Esta función crea una matriz de vista derivada de un punto visual, un punto de referencia indicando el centro de la escena, y un vector ascendente. La matriz mapea la referencia del punto hacia el eje z negativo y el punto visual al origen para que al usar la matriz projection, el centro de la escena permanezca en el centro de la ventana. De forma similar, la dirección descrita por el vector ascendente sobre un plano de vista se mapee en el eje positivo y para que estos puntos asciendan en la ventana. El vector ascendente no tiene por qué ser paralelo a la línea que une la vista del ojo con el punto de referencia.

- Devolver valores double:

```
void glGetDoublev(GLenum pname,GLdouble *params);
```

Esta función devuelve valores para variables de estado simples en OpenGL.

- `‘‘pname = GL_MODELVIEW_MATRIX’’` .- El parámetro params devuelve 16 valores referentes a la matriz modelview que se haya cargado en último lugar, es decir, que esté la primera en todo el conjunto de matrices modelview que se hayan desarrollado.
- `‘‘pname = GL_PROJECTION_MATRIX’’` .- Al igual que en el caso anterior, el parámetro devuelve 16 valores, pero en este caso referentes a la matriz projection, tal y como sucede para el caso anterior.
- `‘‘*params = modelviewmatrix’’` .- Devuelve los valores del parámetro indicado.

- Devolver valores int:

```
void glGetIntegerv(GLenum pname,GLdouble *params);
```

Función que actúa exactamente igual que la función del caso anterior, únicamente varía el tipo de parámetro que devuelve, e este caso se trata de un entero. Esta función devuelve valores para variables de estado simples en OpenGL.

- `‘‘pname = GL_VIEWPORT’’` .- El parametro params devuelve 4 valores: los valores x, y correspondientes a las coordenadas de la pantalla, seguido por su anchura y su altura.
- `‘‘*params = projectionmatrix’’` .- Devuelve los valores del parámetro indicado.

- Mapeo de coordenadas:

```
int gluUnProject(GLdouble winx, GLdouble winy, GLdouble winz, const
    GLdouble modelMatrix[16], const GLdouble projMatrix[16], const
    GLint viewport[4], GLdouble *objx, GLdouble *objy, GLdouble *objz);
```

Esta función mapea unas coordenadas de la ventana especifícas en coordenadas de los objetos modelMatrix, projMatrix y viewport. El resultado se almacena en objx, objy, objz. En nuestro programa estos parámetros toman los siguientes valores:

- ‘‘winx = x’’
- ‘‘winy = (viewport[3] - y)
- ‘‘winz = 1.0
- ‘‘ modelMatrix[16] = modelviewmatrix
- ‘‘projMatrix[16] = projectionmatrix
- ‘‘viewport[4] = viewport
- ‘‘* objx = &(projectedpoint[0])
- ‘‘* *objy = &(projectedpoint[1])
- ‘‘* *objz = &(projectedpoint[2])

- Acción del ratón:

```
void mousedown([nIndex], int nButton, int nShift, int nXCoord, int
    nXCoord);
```

El procedimiento MouseDown se utiliza para especificar acciones que se producen cuando se presiona un determinado botón del ratón. A diferencia de los eventos Click y DblClick, podemos utilizar el evento MouseDown para distinguir entre los botones izquierdo, derecho y central del ratón. También se puede escribir código para crear combinaciones de raztón y teclado que utilicen los modificadores del teclado. Podemos utilizar un evento MouseMove para responder a un evento provocado al mover el ratón. El argumento nButton tiene un significado distinto según se trate de MouseDown y MouseUp, o de MouseMove. Para MouseDown o MouseUp, el argumento nButton indica exactamente un botón por evento; para MouseMove, indica el estado actual de todos los botones.

- ‘‘[nIndex] = nada’’.- Contiene un número que identifica un control de forma única si éste está en una matriz de controles. El parámetro nIndex sólo se transfiere cuando el control forma parte de una matriz de controles.
- ‘‘nButton = button’’.- Contiene un número que especifica qué botón se presiona para desencadenar el evento: 1 (izquierdo), 2 (derecho) o 4 (central).
- ‘‘nShift = state’’.- Contiene un número que especifica el estado de las teclas modificadoras cuando se hace clic con el ratón. Las teclas modificadoras válidas son 1 (MAYÚS), 2(CTRL) y 4(ALT).Si se mantienen presionadas una o más teclas modificadoras mientras se hace clic, el argumento nShift contiene la suma de los valores para las teclas modificadoras. Por ejemplo, si el usuario mantiene presionada la tecla CTRL mientras presiona el botón del ratón, el argumento nShift contiene 2. Pero si el usuario mantiene presionadas las teclas CTRL+ALT mientras presiona el botón del ratón, el argumento nShift contiene 6.
- ‘‘nXCoord = x’’.- Contiene la posición horizontal (nXCoord) actual del puntero del ratón en el formulario.
- ‘‘nYCoord = y’’.- Contiene la posición vertical (nYCoord) actual del puntero del ratón en el formulario.

Estas coordenadas siempre se expresan en términos del sistema especificado de coordenadas del Form y en la unidad de medida que establece el valor de la propiedad ScaleMode.

- Crear la textura:

```
void glBindTexture(GLenum target, GLuint texture);
```

Esta funcion nos permite crear lo que llamamos textura. Llamándola pasando `GL_TEXTURE_2D` y el nombre de una nueva textura creada hacemos que ésta quede unida de la forma que establezcamos con el parametro target. Cuando una textura es ligada a un blanco determinado por el parámetro `target`, la ligadura anterior para ese blanco deja de tener efecto.

- `‘‘target = GL_TEXTURE_2D’’` .- La superficie a la cual la textura está ligada tiene que tener el valor `GL_TEXTURE_1D` o `GL_TEXTURE_2D’’` .
- `‘‘texture = texName` .- El nombre de que asociamos a la textura, `texture` no puede estar en uso en ese momento.

- Interpretación de la textura:

```
void  glTexEnvf(GLenum target, GLenum pname, GLfloat param);
```

El entorno de textura especifica cómo se interpretan los valores de la textura cuando una superficie es matizada. La función textura actúa en una superficie para ser matizada usando el valor de la imagen textura para ésta y produce un color RGBA[7].

- `‘‘target = GL_TEXTURE_ENV’’` .- El entorno de la textura, definido por `GL_TEXTURE_ENV` .
- `‘‘pname = GL_TEXTURE_ENV_MODE’’` .- El nombre simbólico para el parámetro ambiente del valor singular de la textura ha de ser `GL_TEXTURE_ENV_MODE’’` .
- `‘‘param = GL_MODULATE’’` .- Esta constante sólo puede tomar los valores `GL_MODULATE` , `GL_DECAL` , `GL_BLEND` , o `GL_REPLACE` .

- Aplicar una imagen sobre la textura:

```
void  glTexParameteri(GLenum target, GLenum pname, GLint  param);
```

El mapeado de la textura es una técnica que aplica una imagen sobre la superficie de un objeto ya sea una calcamonía o celofán rodeado y reducido. La imagen es creada en un espacio de textura, con un sistema de coordenadas (x,y). Una textura es una imagen uno o dos-dimensional y una serie de parámetros que determinan cómo las muestras derivan de la imagen. La función `glTexParameter` asigna el valor o valores de `param` al parámetro textura especificado en `pname` .

- `‘‘target = GL_TEXTURE_2D’’` .- La textura objetivo que tiene que ser `GL_TEXTURE_1D` o `GL_TEXTURE_2D` .

[7] para la tabla de colores ver https://msdn.microsoft.com/en-us/library/windows/desktop/dd368625(v=vs.85).aspx

- ‘‘pname = GL_TEXTURE_WRAP_S’’.- El nombre simbólico para el valor singular del parámetro textura. Envuelve a los parametros:
  - ‘‘GL_CLAMP’’.- Produce que la coordenada x quede restringida al intervalo [0, 1]. Es útil para prevenir la envoltura cuando mapeamos una imagen sobre un objeto.
  - ‘‘GL_REPEAT’’.- Produce una parte entera de la coordenada x para ser ignorada. OpenGL usa solo una parte fraccional para crear un patrón de repetición. Los elementos de la frontera son accesibles solo si la envoltura es enviada a GL_CLAMP. Incialmente, a GL_TEXTURE_WRAP_S se envía GL_REPEAT.
- ‘‘pname = GL_TEXTURE_WRAP_T’’.- Ocurre la misma acción que en el caso anterior pero, ahora, con la coordenada y.
- ‘‘pname = GL_TEXTURE_MAG_FILTER’’.- La función de aumento de la textura se usa cuando el píxel, siendo del mapeo matizado de un área, es menor o igual que un elemento de la textura. Se envía entonces a la función GL_NEAREST o GL_LINEAR’’.
- ‘‘param = GL_REPEAT’’.- Valor del parametro pname.
- ‘‘param = GL_NEAREST’’.- Valor del parametro pname.

- Obtener imagen y generar “mipmaps”:

```
void gluBuild2DMipmaps(GLenum target, GLint  components, GLint
    width, GLInt  height, GLenum format, GLenum type, const void *
    data);
```

La función obtiene la imagen de entrada y genera todos las imágenes “milmap” así dicha imagen puede ser usada como imagen de textura *milmapeada*.

- ‘‘target = GL_TEXTURE_2D’’.- La textura objetivo, debe ser GL_TEXTURE_2D.
- ‘‘components = GL_RGB’’.- El número del color que compone la textura, que habrá de ser 1, 2, 3 o 4 (en nuestro caso GL_RGB =2).
- ‘‘width = TEXTURE_SIZE’’.- El ancho de la imagen textura.
- ‘‘height = TEXTURE_SIZE’’.- El alto de la imagen textura.
- ‘‘format = GL_RGB’’.- El formato de la información del píxel, habrá de ser uno de entre GL_COLOR_INDEX, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGB, GL_RGBA, GL_BGR_EXT, GL_BGRA_EXT, GL_LUMINANCE, o GL_LUMINANCE_ALPHA.
- ‘‘type = GL_UNSIGNED_BYTE’’.- El tipo de información de data. Tiene que ser uno de los siguientes: GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, o GL_FLOAT.
- ‘‘*data = texture’’.- El puntero de la imagene en la memoria.

- Recuperar entero GLUT:

```
int glutGet(GLenum state);
```

Esta función recupera el valor de la variable state de GLUT representada por enteros. El parámetro state determina qué tipo de estado se devuelve.

- ‘‘state = GLUT_ELAPSED_TIME’’.- Número de milisegundos desde que se llama a la función glutInit.

- Rellamada display:

```
void glutPostRedisplay(void);
```

Marca el plano normal de la ventana actual para ser mostrada nuevamente. En la siguiente iteración, a través de glutMainLoop, vuelve a llamar al muestreo de la ventana que servirá para mostrar, de nuevo, el plano normal de ésta. Múltiples llamadas a glutPostRedisplay antes de la siguiente rellamada de muestreo genera una única rellamada. Esta función puede ser llamada sin el muestreo de la ventana o cubrir la rellamada para remarcar la ventana para su nuevo muestreo.

- Aplicar tonalidad:

```
void glShadeModel(GLenum mode);
```

Las primitivas de OpenGL pueden tener una tonalidad plana o lisa. La tonalidad lisa, como condición base, causa el cálculo del color de los vértices para ser interpolados cuando la primitiva es rasterizada, típicamente se asignan diferentes colores a cada fragmento del píxel resultante. La tonalidad plana selecciona el color calculado de un sólo vértice para todos los píxeles del fragmento generando, por renderización, una única primitiva. En cualquier caso, el color computado de un vértice es el resultado de la iluminación, si está habilitada, o es el color actual en el tiempo en el que es especificado el vértice si la iluminación está deshabilitada. Las tonalidades lisas o planas son indistinguibles para puntos. Aunando vértices y primitivas iniciado por glBegin, cada segmento de textura lisa computa un color entre los vértices i e $i+1$. De forma similar, a cada polígono plano se le da el color del vértice computado, en este caso particular de TRIANGLE_STRIP [8], $i+2$, es el último vértice que especifica el polígono en todos los casos excepto polígonos singulares, donde el primer vértice especifica el color de la tonalidad.

- ‘‘mode = GL_SMOOTH’’.- Valor simbólico que representa la tonalidad.

- Desabilitar graficos OpenGL:

```
void glDisable(GLenum cap);
```

Esta función desabilita capacidades gráficas de OpenGL.

- ‘‘cap = GL_CULL_FACE’’.- Constante simbólica indicando una cualidad de OpenGL. Si está habilitado selecciona una base de polígonos en las coordenadas sinuosas de la pantalla.
- ‘‘cap = GL_DEPTH_TEST’’.- Si está habilitado hace comparaciones profundas y actualiza la intensidad del buffer.

- Asignar modos a los píxeles:

[8] https://msdn.microsoft.com/en-us/library/windows/desktop/dd368583(v=vs.85).aspx

```
void glPixelStorei(GLenum pname, GLint param);
```

Esta función asigna los modos del píxel almacenado que afectan a la operación de la subsecuencia `glDrawPixels` y `glReadPiels` así como deshacer el patrón de los punteros del polígono, los bitmaps y los patrones de textura.

- `‘‘pname = GL_UNPACK_ALIGNMENT’’` .- Nombre simbólico del parámetro para ser establecido. Seis de los parámetros almacenados afectan a cómo la información del píxel es leída por la memoria. En particular, este parámetro establece el alineamiento requerido para empezar la fila de cada píxel (en este caso un byte por alineación).
- `‘‘param = 1’’` .- alor que se le asigna a `pname` .

- Devolver nombres de texturas:

```
void glGenTextures(GLsizei n,GLuint *textures);
```

Esta función devuelve “n“ nombres de texturas del parámetro `textures` . El nombre de la textura no ha de ser necesariamente un número contiguo de enteros, sin embargo, ninguno de los nombres devueltos pueden ser usado inmediatamente antes de llamar a la función `glGenTextures` . Las texturas generadas asumen las dimensiones de la textura objetivo la cual está primeramente ligada mediante la función `glBindTexture` .

- `‘‘n = 1’’` .-Número del nombre de la textura a generar.
- `‘‘*textures = & texName’’` .- Puntero del primer elemento de un array en el que están almacenados los nombres de las texturas generadas

- Habilita gráficos de `OpenGL` :

```
void glEnable(GLenum cap);
```

El comportamiento de esta función ya lo hemos explicado en `glDisable` . Para comprobar utilidad ver página 70.

- `‘‘cap = GL_TEXTURE_2D’’` .- Para este parámetro, si está habilitado se representa una textura 2-dimensional (ver `glDisable` .
- `‘‘cap = GL_LIGHTING’’` .- Si está habilitado, usa la iluminación actual de los parámetros para computar el color del vector o el índice.
- `‘‘cap = GL_LIGHT0’’` .- Si está habilitado, incluye la luz 0 en la evaluación de la ecuación de la iluminación.

- Establecer los parámetros para una fuente de luz:

```
void glLightfv(GLenum light, GLenum  pname, const GLfloat *params);
```

Esta función estable el valor o valores de los parámetros de una fuente de luz individual. El parámetro `light` es un nombre simbólico para `GL_LIGHTi` donde $0 \leq i \leq$ `GL_MAX_LIGHTS` . El parámetro `pname` especifica uno de los parámetros de la fuente de luz. Puede ser

tanto un valor único como un puntero de un array que contenga nuevos valores. Los cálculos de iluminación se habilitan utilizando `glEnable` y `glDisable` con el argumneto `GL_LIGHTING`. Cuando la iluminación está habilitada, las fuentes de luz se habilitan para contribuir a los cálculos de la iluminación. La fuente de luz `i` se activa o desactiva usando `glEnable` y `glDisable` con el argumento `GL_LIGHTi`.

- ``light = GL_LIGHT0'' .- Identificador de la luz. Número de las posibles dependencias de la luz en la implementación pero, como mínimo, son soportadas 8 luces. Éstas están identificadas con el nombre simbólico de `GL_LIGHTi` con `i` de 0 a `GL_MAX_LIGHTS-1`.
- ``pname = GL_AMBIENT'' .- Parámetro de fuente de luz para `light`. En este caso, el parámetro `params` contiene 4 puntos flotantes que especifican el ambiente intensidad RGBA de la luz. Los valores de los puntos flotantes son mapeados directamente. Ni los valores de los enteros ni de los puntos flotantes están agrupados. La intensidad estándar del ambiente luz es (0.0, 0.0, 0.0, 1.0), referentes a los colores "rojo", "verde", "azul" y "transparencia" respectivamente.
- ``pname = GL_DIFFUSE'' .- En este caso ocurre lo mismo que en el anterior salvo que la intensidad estándar de la difusión es (0.0, 0.0, 0.0, 1.0) para las luces distintas de cero, mientras que, para éstas, su difusión es de (1.0, 1.0, 1.0, 1.0).
- ``pname = GL_POSITION .- El parámetro `params` contiene 4 puntos flotantes que especifican la posición de la luz en coordenadas homogéneas del objeto. La posición es trasformada mediante la matriz `modelview` cuando `glLightfv` es llamado (justo cuando hay un punto) y es almacenado en las coordenadas visuales. Si la componente z de la posición es 0.0, la luz es tratada como una dirección fuente. Los cálculos de la iluminación difusa y especular toman la dirección de la luz, pero no de su actual posición, y deshabilitan la atenuación. Por otro lado, los cálculos de la iluminación difusa y especular están basados en la localización actual de las coordenadas de la luz visual y la atenuación cuando está habilitada. De modo que la posición estándar es (0, 0, 1, 0), por consiguiente, la fuente de luz es direccional, paralela al eje z.
- ``*params = lightambient / lightdiffuse / lightpos'' .- Especifica el valor que el parámetro `pname` de la fuente de `ligth` al que va a ser enviado.

- Establecer los parámetros para un modelo de iluminación:

```
void glLightModeli(GLenum pname, GLint param);
```

Esta función establece el parámetro del modelo de iluminación. El nombre del parámetro `pname` denomina al parámetro y, `param`, asigna el nuevo valor o valores de los parámetros de la fuente individual. En modo RGBA, el color iluminado de un vértice es la suma de las emisiones de intensidad del material, el producto del reflejo del ambiente del material y la intensidad de la iluminación del ambiente de la escena completa, y la contribución de cada fuente de luz habilitada. Cada fuente de luz contribuye con la suma de 3 términos: el ambiente, la difusión y la reflexión especular.

⋙ La contribución del ambiente de la fuente de luz es el producto de la reflexión del ambiente del material y la intensidad del ambiente.

⋙ La contribución de la difusión la de fuente de luz es el producto de la reflexión de la difusión del material, la intensidad de la difusión de la luz, y el producto escalar de la normal del vértice con el vector normalizado de la dirección de la fuente de dicha luz.

⋙ La contribución del especular de la fuente de luz es el producto del especular reflectante del material, la intensidad especular de la luz, y el producto vectorial normalizado del vector que une el vértice con el punto de vista y del vector que une el vértice con la luz, incrementando la fuerza del brillo en el material.

Las 3 contribuciones de las fuentes de luz son atenuadas por igual basándose en la distancia desde el vértice a la fuente de luz, y en la dirección de la fuente de luz. Todos los productos escalares son reemplazados por 0 cuando el resultado de la evaluación es un valor negativo. La componente "alpha" del resultado del color iluminado es establecido por el valor `alpha` de la difusión reflectante del material.

- `''pname = GL_LIGHT_MODEL_TWO_SIDE''` .- Valor singular del modelo del parámetro iluminación. En este caso, el parámetro `param` es un valor singular entero que especifica si los cálculos de iluminación han sido hechos para polígonos. No tiene efecto sobre los cálculos realizados para la iluminación de los puntos, lineas o bitmaps. Si `param` es 0 la iluminación se especifica por un lateral y solamente los parámetros del frente del material son utilizados en la ecuación. De otra forma, se especifica la iluminación por los dos lados. En este caso, la parte opuesta de los polígonos es iluminada usando los parámetros para la parte de detrás del material, y tiene reservada sus normales antes de que la ecuación de la iluminación sea evaluada. Los vértices de la cara frontal de los polígonos siempre están iluminados usando los parámetros del material frontal que no cambian sus normales. Por defecto, éstos son 0.
- `''param = 1''` .- El valor al cual `param` va a ser asignado.

- Asignar valores a los parámetros del material:

```
void glMaterialfv(GLenum face, GLenum pname, const GLfloat *params);
```

Esta función asigna valores a los parámetros del material. Hay dos combinaciones establecidas de parámetros de material. Una, la cara frontal, se usa para sombrear los puntos, líneas, bitmaps y todos los polígonos (cuando la iluminación por ambos lados está desactivada) o los polígonos de cara frontal (si la iluminación por ambos lados está activada). La otra, la cara de detrás, se usa para la cara trasera de los polígonos sólo cuando la iluminación por ambos lados está activada. Esta función coge 3 argumentos, con el que nosotros consideramos el parámetro `params` se especifica qué valor va a ser asignado al parámetro específico. Los parámetros de los materiales se utilizan en la ecuación de la luminosidad, que es aplicada opcionalmente a cada vértice. La ecuación es tratada en `glLightModel` . Los parámetros del material pueden ser actualizados todo el tiempo. En particular, `glMaterial` puede ser llamado entre `glBegin` y su correspondiente `glEnd` . Cuando sólo hay un único parámetro de material por vértice para ser cambiado es preferible, en ese caso, llamar a la función `glColorMaterial` .

- `''face = GL_FRONT_AND_BACK''` .- Variable referida a la cara o caras que van a ser actualizadas. Será una de entre `GL_FRONT` , `GL_BACK` , `GL_FRONT` o `GL_BACK` .
- `''pname = GL_AMBIENT_AND_DIFFUSE''` .- Variable referida al parámetro material de la cara o caras que serán actualizadas. En este caso, dicho parámetro equivale a llamar a la función `glMaterial` dos veces con los mismos valores del parámetro, una vez con `GL_AMBIENT` y otra con `GL_DIFFUSE` . En la primera llamada, el parámetro `params` contiene 4 valores de puntos flotantes que especifican la reflexión del ambiente RGBA del material. Los valores enteros son mapeados linealmente entre -1.0 y 1.0. Los valores

de los puntos flotantes son mapeados directamente y, ni estos ni los enteros son fijados. El ambiente reflectante para ambas caras es, por defecto, (0.2, 0.2, 0.2, 1.0). En la segunda ocurre lo mismo que en el anterior salvo que, por defecto, la reflexión de la difusión es (0.8, 0.8, 0.8, 1.0).

- `‘‘*params = material’’` .- Valor que asigna el parámetro `GL_SHININESS` .

- Establecer área de plano de bits:

```
void glClear(GLbitfield mask);
```

Esta función establece el área de plano de bits de la ventana para los valores previamente seleccionados mediante `glClearColor` , `glClearIndex` , `glClearDepth` , `glClearStencil` y `glClearAccum` . Es posible limpiar varios buffer de color simultáneamente seleccionando más de un buffer al mismo tiempo a través del comando `glDrawBuffer` . Esta función coge un argumento único, cuyo valor se opera bit a bit, indicando qué buffer habrá de ser limpiado. El valor para el cual cada buffer se limpia depende del valor de limpieza establecido por dicho buffer. Si éste no está presente, la llamada no tiene efecto.

- `‘‘mask = GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT’’` .- Operadores de máscaras *“OR”* bit a bit que indican el buffer para ser borrado. En este caso, con el primer comando, el buffer está activado en ese momento para escribir los colores. Con la segunda, se profundiza en el buffer.

- Establecer color:

```
void WINAPI glColor3f(GLfloat red, GLfloat green, GLfloat blue);
```

`GL` guarda dos elementos, un índice para el valor singular actual del color y 4 valores RGBA para el color actual. La función establece 4 nuevos valores RGBA. Esta función tiene dos grandes variantes: `glcolor3` y `glcolor4` . `Glcolor3` especifica las nuevas variantes de los valores rojo, verde, azul y establece el alpha actual en 1.0 (a plena intensidad) de forma implícita. `Glcolor4` especifica las 4 componentes de forma explícita. Los valores del color actual están almacenados en un formato de puntos flotantes sin mantisa especificada ni tamaño de los exponentes. Un componente independiente de color, cuando se especifica, está linealmente mapeado con los valores de los puntos flotantes tan largo como sea su valor de mapeado representado, siendo 1.0 plena intensidad y 0.0, ninguna intensidad. Los componentes enteros dependientes, al igual que los anteriores, son linealmente mapeados pero, ahora, entre el -1.0 y el 1.0.

En nuestro caso todos los parámetros toman el mismo valor, 1.0.

- Trasladar posiciones:

```
void glTranslatef(GLfloat x, GLfloat y, GLfloat z);
```

Esta función realiza una traslación especificada por (x,y,z). El vector traslación suele ser computado por una matriz de traslación 4x4 [(1,0,0,x),(0,1,0,y),(0,0,1,z),(0,0,0,1)]. La matriz actual (ver `glMatrixMode` más abajo) es multiplicada por esta matriz de traslación reemplazando la matriz actual por el producto resultante. Si el modelo de matriz

es, o bien `GL_MODELVIEW`, o bien `GL_PROJECTION`, todos los objetos dibujados después de `glTranslatef` son llamados trasladados. Se utiliza `glPushMatrix` y `glPopMatrix` para guardar y restaurar el sistema de coordenadas no trasladadas, respectivamente.

- Renderización de la esfera:

```
void glutSolidSphere(GLdouble radius, GLint slices, GLint stacks);
```

Renderiza una esfera centrada en las coordenadas originales del modelo con un radio específico. La esfera es, además, subdividida alrededor del eje z en tiras y en pilas a lo largo del eje z.

- `‘‘radius = sphere[3]’’`.- Radio de la esfera
- `‘‘slices = 16’’`.- Número de subdivisiones alrededor del eje z (similar a las líneas de longitud).
- `‘‘stacks = 16’’`.- Número de subdivisiones a lo largo del eje z (similar a las líneas de latitud).

- Cambio de buffer:

```
void glutSwapBuffers(void);
```

Representa un cambio de buffer de la capa en uso a la ventana actual. Específicamente, la función promueve los contenidos del buffer trasero de la capa en uso para convertirlos en contenidos frontales del buffer. Los contenidos traseros se convierten, entonces, en indefinidos. La actualización usual tiene lugar durante el retroceso vertical del monitor justo en el momento inmediatamente posterior a que la función sea llamada. Además, se realiza de forma implícita la función `glFlush` antes de acabar. Por otro lado los comandos de subsercuencia de `OpenGL` pueden ser emitidos después de llamar a esta función pero no serán ejecutados hasta que el cambio de buffer esté completo. Si la capa actual no está almacenada, esta función no tiene efecto.

- Especificación de transformación afín:

```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

Esta función espefica la transformación afín (x,y) de las coordenadas de los dispositivos normalizados a las coordenadas de la ventana. Permite a (x_nd, y_nd) ser las coordenadas normalizadas de los dispositivos. Las coordenadas de la ventana son computadas de la siguiente forma:

$$\begin{aligned} x\_w &= (x\_nd + 1)(width/2) + x \\ y\_w &= (y\_nd + 1)(width/2) + y \end{aligned}$$

La anchura y altura de la ventana son asintóticamente restringidas al rango del que dependa la implementación. Este rango es requerido llamando a `glGet` con el argumento `GL_MAX_VIEWPORT_DIMS`.

- Estado de matrices de renderización:

```
void glMatrixMode( GLenum mode);
```

Esta función establece el estado de las matrices actuales de ejecución.

- `‘‘mode = GL_PROJECTION’’` .- Aplica operaciones de subsecuencias de matrices al conjunto de matrices projection.
- `‘‘mode = GL_MODELVIEW’’` .- Aplica operaciones de subsecuencias de matrices al conjunto de matrices modelview.

- Establecer espacio de visualización de la escena:

```
void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear,
     GLdouble zFar);
```

Esta función especifica el tronco de visualización sobre el sistema de coordenadas del “Mundo Real”. En general, la relación de aspecto en `gluPerspective` debe combinar la relación de aspecto de la ventana asociada. Por ejemplo, `aspect` $= 2.0$ quiere decir que el ángulo de la vista es 2 veces en x lo que es en y. Si la ventana es el doble de ancho de lo que lo es de alto, se muestra la imagen sin distorsión. La matriz generada por `gluPerspective` es multiplicada por la matriz actual, justo como si la función `glMultMatrix` fuera llamada con la matriz generada. Para cargar la matriz de perspectiva sobre el conjunto de matrices actuales en su lugar, debe de preceder a la llamada la función `glLoadIdentity` .

- `‘‘fovy = 60’’` .- Área del ángulo vista, en grados, en la dirección y.
- `‘‘aspect = (GLfloat)w / (GLfloat)h’’` .- Relación de aspecto que determina el área de la vista en la dirección x. Dicha relación se establece del ancho de x a la altura de y.
- `‘‘zNear = 0.1’’` .- Distancia del observador al plano de delimitación más cercano.
- `‘‘zFar = 100.0’’` .- Distancia del observador al plano de delimitación más lejano.

- Inicialización de las librerías de `GLUT` :

```
void glutInit(int *argcp, char **argv);
```

Esta función inicializa las librerías de `GLUT` y guarda una sesión con el sistema de la ventana. Durante este proceso, `glutInit` puede causar el término del programa `GLUT` con un mensaje de error al usuario si `GLUT` no ha podido ser correctamente inicializado. Este tipo de situaciones pueden darse por fallos al conectar con el sistema de la ventana, que el sistema no soporte `OpenGL` u opciones de de líneas de código no válidas.

- `‘‘*argcp = &argc’’` .- Un puntero a la variable `argc` no modificada del programa principal `main` . Tras el `return` , el valor del puntero será modificado a través de `argcp` ya que `glutInit` extrae las opciones de las líneas de comando resueltas por la librería `GLUT` .
- `‘‘**argv = argv’’` .- Variable `argv` no modificada del programa principal `main` . Como ocurría con `argcp` , la información de `argv` será actualizada del mismo modo.

- Inicialización de la visualización del renderizado:

```
void glutInitDisplayMode(unsigned int mode);
```

Este modo de inicialización de la visualización se usa cuando se crea un nivel superior de la ventana, subventanas o una transparencia de un determinado modo de visualización de `OpenGL` para ser creado.

- ``mode = GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH'' .- Modo de visualización. Normalmente la comprobación bit a bit mediante *"OR"* genera un modo de mapa de bits.
  - ⋙ `GLUT_DOUBLE` .- Máscara de bits para seleccionar los double de la ventana. Esto anula `GLUT_SINGLE` si también se especifica.
  - ⋙ `GLUT_RGB` .- Máscara de bits para seleccionar un modo RGBA de la ventana. Éste se pone por defecto si ni `GLUT_RGBA` ni `GLUT_INDEX` son especificados.
  - ⋙ `GLUT_DEPTH` .- Máscara de bits para seleccionar una ventana con un almacenamiento profundo.

- Crear una nueva ventana de visualización:

```
int glutCreateWindow(char *name);
```

Esta función crea una ventana de nivel superior. El nombre debe estar previsto para el sistema de la ventana como nombre de la ventana. El propósito es que el sistema de la ventana marcará la ventana con ese nombre, implícitamente, la ventana actual es establecida como la reciente ventana creada. Cada ventana creada tiene asociado un único contexto `OpenGL` . El estado cambia el contexto de `OpenGL` asociado a la ventana cuando puede hacerlo inmediatamente después de que la ventana se cree. El estado de visualización se produce, inicialmente, para que la ventana sea mostrada pero, el estado actual de la ventana no entrará en acción hasta que la función `glutMainLoop` sea introducida. Esto quiere decir que, hasta que `glutMainLoop` no se llame y se renderice para crear la ventana es inefectiva, ya que la ventana no podrá ser visualizada todavía. Por otro lado se devuelve un pequeño entero que identifica la ventana. El rango que identifica la asignación empieza en 1. Este identificador puede ser usado cuando llamamos a la función `glutSetWindow` .

- ``*name = argv[0]'' .- Secuencia de caracteres ASCII para utilizar como nombre de la ventana.

- Crear menú de interacción:

```
int glutCreateMenu(void (*func)(int value));
```

Esta función crea un nuevo menú troquelado y devuelve un pequeño entero como identificador. El rango que identifica la asignación empieza en 1. El rango identificador del menú es independiente del rango del identificador de la ventana. Cuando la retrollamada del menú es realizada porque se selecciona un acceso del menú desde éste, el menú actual será implícitamente establecido antes de que la llamada sea efectuada.

- ``(*func)(int value) = menu'' .- La retrollamada de la función para el menú que se realiza cuando se selecciona un acceso del menú desde éste. El valor pasado a la retrollamada es determiando por el valor que se selecciona en el menú de acceso.

- Añadir menú de accesos:

```
void glutAddMenuEntry(char *name, int value);
```

Añade al menú de accesos una nueva entrada al final del actual. El string `name` será visualizado por la entrada más reciente del menú. Si la entrada es seleccionada por el usuario, la retrollamada del menú será realizada pasando `value` como parámetro de la llamada.

- ``*name'' .- Cadena de caracteres ASCII para visualizarlos como acceso del menú.
- ``value'' .- Valor que devuelve la retrollamada del menú si el acceso es seleccionado.

- Vinculación de los botones del ratón:

```
void glutAttachMenu(int button);
```

Esta función vincula un botón del ratón con la ventana con el fin de identificar el menú actual. Mediante la vinculación del identificador del menú a un botón, el nombre del menú aparecerá cuando el usuario presione el botón especificado. El parámetro `button` tomará uno de los siguientes valores: `GLUT_LEFT_BUTTON` , `GLUT_MIDDLE_BUTTON` , o `GLUT_RIGHT_BUTTON` .

- ``button = GLUT_RIGHT_BUTTON'' .- Botón al que se le atribuye el menú.

- Sincronización de la renderización:

```
void glutIdleFunc(void (*func)(void));
```

Esta función establece la sincronización de la llamada de las funciones de forma que el programa `GLUT` puede realizar tareas de procesamiento en segundo plano o animación continua cuando el sistema de la ventana no se está recibiendo eventos. Si está habilitado, la retrollamada de sincronización es continuamente llamada cuando no se están recibiendo eventos. La rutina de la retrollamada no tiene parámetros. La ventana actual y el menú actual no son alterados antes de una llamada. Los programas con ventanas múltiples y/o menús deben establecer explícitamente la ventana y/o menú actuales, y no sus dependencias con el marco actual. La cantidad de cálculos y renderizaciones realizadas en una sincronización de retrollamada debe ser minimizada para evitar que afecte a la respuesta interactiva del programa. En general, no más de un frame de renderización debe ser hecho en una sincronización. Pasando NULL a la `glutIdleFunc` desabilitamos la generación éstas.

- ``(*func)(void) = idle'' .- La nueva retrollamada de renderización.

- Visualización de las renderizaciones:

```
void glutDisplayFunc(void (*func)(void));
```

Esa función establece la retrollamada de la visualización para la ventana actual. Cuando `GLUT` determina que el plano normal para la ventana necesita ser visualizado de nuevo, se llama a la retrollamada de visualización para la ventana. Antes de la retrollamada, la

ventana actual se establece en la ventana que necesita ser visualizada de nuevo y, si no se ha registrado ningún recubrimiento de retrollamada de visualización, la "capa en uso" se establece en el plano normal. La retrollamada de visualización se llama sin parámetros. La región completa del plano normal debe ser mostrada de nuevo en respuesta a ésta (esto incluye a los buffer secundarios si nuestros programas dependen de sus estados). `GLUT` determina cuándo la retrollamada de visualización debe ser activada basándose en el estado de la ventana al mostrarse de nuevo. El estado de la revisualización de la ventana puede ser establecido de forma explícita llamando a la función `glutPostRedisplay`, o implícitamente como resultado del daño reportado por el sistema de la ventana. Actualizar múltiples revisionados para la ventana pueden unirse mediante `GLUT` para minimizar el número de llamadas a la estas retrollamadas.

Cuando una capa es establecida en una ventana, pero no hay registrada una retrollamada de visualización de la capa, la retrollamada es usada para revisualizar ambos: la capa y el plano normal. En este caso, la çapa en uso"no se cambia implícitamente en la entrada de la retrollamada de visualización. Para entender cuántas retrollamadas distintas para las capas y el plano normal pueden ser establecidas podemos ver la función `glutOverlayDisplayFunc` en el enlace de referencia. Cuando una ventana es creada no existe ninguna retrollamada de visualización, es responsabilidad del programador instalar una retrollamada para la ventana antes de que ésta sea mostrada. Una retrollamada de visualización debe ser registrada para todas las ventanas que sean mostradas. Si una ventana se muestra sin una retrollamada de visualización registrada producirá un *"error irreparable"*. En cuanto al `''return''`, el estado de daño normal de la ventana (devuelto llamando a `glutLayerGet(GLUT_NORMAL_DAMAGED)`) se limpia. Si no hay registrada ninguna retrollamada de la capa, el estado de la capa de daño de la ventana (devuelta llamando a `glutLayerGet(GLUT_OVERLAY_DAMAGED)`) se limpia también.

- `''(*func)(void) = display''`.- Retrollamada para la nueva visualización.

- Remodelado de la escena:

```
void glutReshapeFunc(void (*func)(int width, int height));
```

Esta función establece la retrollamada del modelao para la ventana actual. La retrollamada de modelado se activa cuando una nueva ventana se forma de nuevo. Una retrollamada de modelado se activa también inmediatamente antes de la primera retrollamada de visualización, después de que una ventana sea creada o se establezca una capa en cualquier lugar de la ventana. Los parámetros `width` y `height` de la retrollamada especifican el tamaño de la nueva ventana en píxeles. Antes de la retrollamada la ventana actual es establecida en la ventana que ha sido remodelizada. Si una retrollamada de remodelado se registra en una ventana, o se pasa NULL a la función `glutReshapeFunc`, se usará la retrollamada de remodelado con sus valores por defecto, es decir, simplemente se llamará a la función `glViewport(0,0,width,height)` en el plano normal (y en las capas si existe alguna). Si una capa es establecida para una ventana, también se genera una retrollamada de remodelado. El papel de esta retrollamada es el de actualizar el plano normal y las capas de la ventana. Cuando un nivel superior de la ventana se remodela, las subventanas no lo harán. Corresponde al programa `GLUT` dirigir el tamaño y la posición de las subventanas sin un nivel superior. Aún así, las retrollamadas de remodelado serán activadas para éstas cuando su tamaño cambie al llamar a `glutReshapeWindow`.

- `''(*func)(int width, int height) = reshape''`.- Retrollamada del nuevo modelado.

- Interacción con el teclado:

```
void glutKeyboardFunc(void(*func)(unsigned char key, int x, int y))
    ;
```

Esta función establece la retrollamada del teclado para la ventana actual. Cuando un usuario teclea sobre la pantalla, cada tecla presionada genera un carácter ASCII que generará una retrollamada de teclado. El parámetro de retrollamada `key` es el carácter ASCII generado. El estado de caracteres modificados como el caso *"Shift"* no puede ser determinado directamente, sólo tendrá efecto en la base de datos de ASCII devuelta. Los parámetros de la retrollamada x e y indican la posición del ratón en las coordenadas relativas de la ventana cuando el ratón es presionado. Cuando una nueva ventana es creada, no hay registros iniciales de las retrollamadas del teclado, y las pulsaciones de claves ASCII en la ventana son ignoradas. Pasando NULL a `glutKeyboardFunc` deshabilitamos la generación de retrollamadas de teclado. Durante la retrollamada de teclado, la función `glutGetModifiers` puede ser llamada para determinar el estado de las claves modificadas cuando ocurre una pulsación, generando una retrollamada.

- `‘‘(*func)(unsigned char key, int x, int y) = keyboard’’` .- La nueva retrollamada del teclado.

- Interacción con el ratón en movimiento:

```
void glutMotionFunc(void(*func)(int x, int y));
```

Esta función establece la retrollamada de movimiento respecto de la ventana actual. La retrollamada de movimiento para una ventana se llama cuando el ratón se mueve en la ventana mientras uno o varios botones están siendo presionados. Los parámetros x e y de la retrollamada indican la posición de las coordenadas relativas del ratón en la ventana. Pasando NULL a `glutMotionFunc` deshabilitamos la generación de retrollamadas de movimiento.

- `‘‘(*func)(int x, int y) = mousemove’’` .- La nueva retrollamada de la función de movimiento.

- Interacción con el ratón estático:

```
void glutMouseFunc(void (*func)(int button, int state, int x, int y
    ));
```

Esta función establece la retrollamada del ratón para la ventana actual. Cuando un usuario presiona y suelta un botón del ratón en la ventana, cada presión y cada vez que se suelta dicho botón se genera una retrollamada de ratón. El parámetro `button` puede ser uno de entre `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON`, o `GLUT_RIGHT_BUTTON`. Sin embargo, si trabajamos con un sistema de sólo 2 botones de ratón, no será posible generar la retrollamda de `GLUT_MIDDLE_BUTTON`. El parámetro `state` es, o bien `GLUT_UP`, o bien `GLUT_DOWN`, indicando si la retrollamada fue producida por soltar o por pulsar un botón, respectivamente. Los parámetros de retrollamada x e y indican las coordenadas relativas de la ventana cuando el estado del botón del ratón cambia. Si se activa la retrollamada `GLUT_DOWN` para un botón específico, el programa puede asumir que la retrollamada de `GLUT_UP` para el mismo botón será generada (asumiendo que la ventana seguía teniendo

una retrollamada de ratón registrada) cuando el botón del ratón se suelte, aunque el ratón se haya movido fuera de la ventana. Si un menú está vinculado a un botón en una ventana, las retrollamadas del ratón no se generarán para ese botón. Durante la retrollamada de ratón, la función `glutGetModifiers` puede ser llamada para determinar el estado de de las claves modificadas cuando ocurra un evento de ratón, generando, así, la retrollamda. Pasando NULL a `glutMouseFunc` deshabilitamos la generación de retrollamadas de ratón.

- `‘‘(*func)(int button, int state, int x, int y) = mousedown’’` .- La nueva retrollamada de la función de movimiento.

- Proceso en bucle de las funciones de `GLUT` :

```
void glutMainLoop(void);
```

Esta función hace entrar a `GLUT` en un proceso de bucle. Esta rutina debe ser llamada como máximo una vez en un programa de `GLUT` . Una vez llamada, esta rutina no volverá a inicializarse,e llamará tantas veces como se necesiten según las retrollamadas que se hayan registrado automáticamente.