



Simulador de Helicoptero

PROYECTO DE FINAL DE CARRERA

INGENIERIA INFORMATICA

Jesús de Santos García

Andrés Ruiz Flores

INDICE

1. Introducción
2. Descripción del proyecto
 1. Descripción
 2. Versiones
 3. Manual de usuario
3. Física del helicóptero
 1. Ecuaciones básicas para los cálculos más importantes
 2. Base técnica del helicóptero
4. Implementación Gráfica
 1. DirectX
 2. Transformaciones geométricas
 3. Sistema de cámaras
 4. Grafo de escena
 5. Representación del cielo
 6. Sistema de Flares
 7. Descripción del algoritmo empleado para renderizar el terreno
5. Implementación del modelo físico
 1. Ecuaciones diferenciales
 2. Simulación de sólido rígido
 3. Descripción de un modelo físico de un helicóptero
6. Descripción de las posibles mejoras
 1. Sistema Multiplayer
 - i. Arquitecturas de diseño de un sistema de juego por red
 - ii. Modelo teórico de un sistema cliente-servidor
 - iii. Sistemas basados en Winsock
7. Relación de clases
8. Bibliografía
- A. Apéndices
 - a. Descripción de la primera aproximación a un objeto sobre un terreno

1. INTRODUCCIÓN

Este trabajo es la documentación del proyecto de fin de carrera de Ingeniería Informática. Se presentó en la asignatura Sistemas Informáticos el año 2001.

Los autores de esta práctica son Andrés Ruiz Flores y Jesús de Santos García. El profesor que llevo este proyecto fue Fernando Sáenz Pérez.

El trabajo fue realizado en dos grandes partes. La primera gran parte se centro en la parte gráfica del proyecto, mientras que la segunda fue la que llevo todo el desarrollo del modelo físico.

2. DESCRIPCIÓN DEL PROYECTO

1. Descripción

El proyecto descrito en estas páginas es un simulador de helicóptero. El entorno de la simulación es un paisaje montañoso con suficientes dimensiones como para permitir unas horas de vuelo.

Para el renderizado del paisaje se emplean técnicas de reducción de detalle que permiten visualizar un paisaje con horizonte infinito. Para la simulación del comportamiento del helicóptero se ha empleado un modelo físico que funciona en tiempo real.

2. Historial de Versiones

18/12/2000 Versión v0.0

Versión preliminar al proyecto.
Ejecutable a parte donde se hacen algunas pruebas con transformaciones 3d sobre una tarjeta grafica.

Esta entrega está descrita en el **apéndice A**.

11/01/2001 Versión v0.10

Primera versión del proyecto.
Se definen las primeras implementaciones casi definitivas de Point3, Matrix y Camera.
Se muestra en una prueba rápida un pseudo terreno y un helicóptero importado del programa 3DStudioMax.
Este ultimo paso habrá que hacerlo de un modo mucho mas estudiado diseñando posiblemente un modelo de objeto propio y un exportador.

--/01/2001 Versión v0.11

Algunos bugs corregidos en la inicialización de DirectX 8.0.
Incorporación del sistema de generación automática de código con DOC++

23/02/2001 Versión v0.30

Incorporación de numerosas mejoras a nivel gráfico.

- SkyBoxes
- Render de terrenos con Octree/LOD
- Lectura del formato .X para el helicóptero
- Definición del grafo de escena

24/04/2001 Versión v0.45

Implementación de FollowCamera
Mejora en la definición del grafo de escena.

- Hierarchically culling con BoundingBoxes
- HitTesting mediante rayos

Sistema de flare. Posiblemente el método de representar el cielo deba ser mejorado.

Primera implementación de un modelo físico totalmente simplificado

15/05/2001 Versión v0.60

Mejorada la implementación de FollowCamera.
Implementación de un nuevo sistema de cielo (SkyDome)
El Sol ahora se pinta además del flare como un circulito.

Segunda implementación del sistema físico. Implementada una versión de integrador ODE mediante el método de Euler.
Hemos pasado de trabajar con matrices de orientación en el sólido rígido a cuaterniones que son más eficientes.

Extensión del D3DWRapper con el concepto de Shaders. Creación de los gestores dinámicos de Buffers dinámicos.

20/06/2001 Versión v0.80

Acabada la implementación del modelo Físico. Pero el control sigue siendo algo difícil. Sería necesaria la implementación de algún mecanismo de control automático o algo similar.

Implementación de un generador automático de texturas y de un generador de sombras para el terreno (TerrainLayer).

Diseñada e implementada la clase para escribir a disco TGAs.

Diseño e implementación de un gestor de cambios de estados de la tarjeta gráfica. Ahora cacheamos todos los cambios de estado, antes de pasarlos a Dx. También hemos incorporado LazyStates.

10/07/2001 Versión v0.90

Implementación de la interface de usuario. Muy rudimentaria, pero mejor que nada.

Incorporación de sombreado al helicóptero. No hemos conseguido encontrar un modelo de helicóptero correctamente texturado, así que nos quedaremos con este definitivamente.

Mejoras en la integración del modelo físico con el escenario.

16/07/2001 Versión v0.95

Mejorado el sistema de render de terrenos. Ahora todos los triángulos se empaquetan en llamadas optimizadas.

Realizadas algunas optimizaciones en el engine gráfica para tarjetas que no aceleren la transformación por hardware.

Incorporación al modelo físico de algunos mecanismos de control automático que hacen mas sencillo manejar el helicóptero.

3. Manual de usuario

Requisitos

Mínimos

Windows 95, Windows 98, Me, 2000

DirectX v8.0

Pentium II 350 Mhz

64 Mb de Ram

Tarjeta aceleradora de primera generación (Voodoo, TNT)

Recomendados

Windows ME

DirectX v8.0

Pentium III 500 Mhz

128 Mb de Ram

Tarjeta aceleradora de segunda generación (GeForce)

Instalación

El programa no necesita una instalación especial. Se debe mantener la estructura de directorios que se encuentra en el CD original. Manteniendo esa estructura, se puede copiar en cualquier parte del disco duro.

Ejecución

El ejecutable del programa es *Simulador.exe*

Una vez ejecutado, las teclas disponibles son las siguientes.

- **Espacio:** con esta tecla sacamos la consola de mensajes del programa. Al pulsar otra vez espacio la consola se esconde. En la consola se muestran información que puede ser útil para depurar el programa o para averiguar las razones de un funcionamiento incorrecto. Con las teclas **RePag** y **AvPag** se puede navegar hacia arriba y hacia abajo en la consola.
- **F1:** Activamos el modo gráfico en ventana

- **F2:** Activamos el modo gráfico en pantalla completa
- **Cursores:** Con las teclas de los cursores arriba y abajo podemos alejar / acercar la cámara del punto de vista del helicóptero.
- **Ratón:** Con el ratón podemos mover la cámara para cambiar el punto de vista del helicóptero.
- **W / S:** Con estas teclas aceleramos / deceleramos el motor del helicóptero. Esta aceleración se traduce directamente en la velocidad de giro del rotor principal, que es fuerza principal que mueve el helicóptero. La velocidad actual de helicóptero se muestra visualmente en una barra inferior.
- **Q / A / O / P:** Con estas teclas movemos hacia arriba, abajo, izquierda y derecha la palanca del cíclico. La situación de la palanca del cíclico se representa gráficamente mediante un círculo en la esquina inferior derecha. Cuando soltamos alguna de estas teclas, la palanca vuelve automáticamente a su sitio. En posteriores versiones del simulador esta palanca debería mapearse a un joystick para facilitar el uso.
- **Z / X:** Con estas teclas aceleramos/desaceleramos la velocidad del rotor trasero. Al desfasarse este rotor con respecto al principal, se producen fuerzas laterales que hacen que el helicóptero gire.



3. FÍSICA DEL HELICÓPTERO

Ecuaciones básicas para los cálculos más importantes en el diseño de un helicóptero

Síntesis

Objetivo

Sintetizar en poco espacio una serie de ecuaciones de fácil acceso a las que deberemos acudir en varias ocasiones

Descripción

A continuación se pasan a exponer, de manera concisa, las principales ecuaciones físicas que vamos a necesitar a lo largo del diseño del helicóptero, así como una pequeña descripción de ellas.

Estos son, por así decirlo, los parámetros que definen fundamentalmente un modelo.

Desarrollo

Superficie de giro del rotor

Esta es la superficie que cubren las palas del rotor en una revolución

$$F = \frac{D^2 \times \Pi}{4}, \text{ donde } D \text{ es el diámetro del rotor.}$$

Se mide en metros cuadrados.

Carga por superficie de giro

Esta es la relación entre el peso del modelo y la superficie de giro.

$$R = \frac{G}{F}, \text{ donde } G \text{ es el peso del modelo y } F \text{ la superficie de giro del rotor.}$$

Se mide en $\frac{kp}{m^2}$.

Densidad de superficie

Esta es la relación entre las superficies superiores de todas las palas que giran en un sistema del rotor y la superficie de giro del rotor.

$C = \frac{Z \times f1 \times 100}{F}$, donde Z es el numero de palas del rotor, f1 la superficie de una pala individual en m² y F la superficie de giro del rotor también en m². Este valor es un porcentaje.

Carga de la pala

Esta es la relación total del modelo y la superficie de las palas del rotor.

$B = \frac{G}{Z \times f1}$, donde G es el peso del modelo, Z el número de palas del rotor y f1 la superficie de una pala individual del rotor en metros cuadrados.

Velocidad de giro

Esta es la velocidad de un punto que gira en el sistema del rotor. Según la distancia de este al eje de rotación, recorrerá un trayecto mayor o menor en cada revolución.

$V = \frac{2 \times r \times \Pi \times n}{60}$, donde r es la distancia del punto al eje de rotación, y n las revoluciones del motor.

Se mide en metros por segundo.

Fuerza centrífuga

En este caso se entiende como fuerza centrífuga la fuerza radial que actúa desde el eje de rotación hacia el exterior producida por el peso de la pala que gira en torno al eje del rotor. Esta será la fuerza que carga las conexiones de las palas del rotor, articulaciones de inclinación de la pala...

$P = \frac{G \times V_s^2}{981 \times r_s}$, donde G es el peso de la pala, r_s la distancia del centro de gravedad al eje de rotacion y V_s la velocidad de giro del centro de gravedad de la pala.

Este valor se mide en kilopondios.

Grado de potencia L

Esta es la relacion entre la potencia del motor y el peso en vuelo. Esto indica, por tanto, la potencia necesaria para elevar un kilogramo de peso del modelo. Esto, en la realidad, suele ser poco util, porque los fabricantes solo proporcionan datos de la potencia punta que alcanzan sus motores.

$L = \frac{N}{G}$, donde N es la potencia del motor y G es el peso en vuelo.

Esto se calcula en caballos por kilopondio.

Suele ser más útil el calculo sobre la cilindrada del motor, que da una idea más aproximada para cada motor, aunque también los motores de igual cilindrada pueden alcanzar distintos rendimientos.

$L = \frac{H}{G}$, donde H es la cilindrada del modelo en centímetros cúbicos.

Esto se calcula en centímetros cúbicos por kilopondio.

Base técnica del helicóptero

Síntesis

Objetivo

Presentar un pequeño resumen sobre los contenidos del documento “Bases de la técnica del helicóptero” referente a los principales puntos a tener en cuenta a la hora de diseñar un simulador de helicópteros

Descripción

Este documento trata de los aspectos principales que se deben observar en el desarrollo de la simulación de un helicóptero.

Desarrollo

Propulsión del rotor

Dentro de los distintos tipos de propulsión que puede tener un helicóptero encontramos:

Con hélices: Situadas en las palas del rotor, estas hélices proporcionan la tracción horizontal que ponen en movimiento el rotor, aunque tiene problemas evidentes debido a la colocación de los motores en las palas.

Por aire comprimido: Un compresor expulsa el aire comprimido por cada pala. Tiene el problema de construcción de unas palas demasiado gruesas para el paso del aire

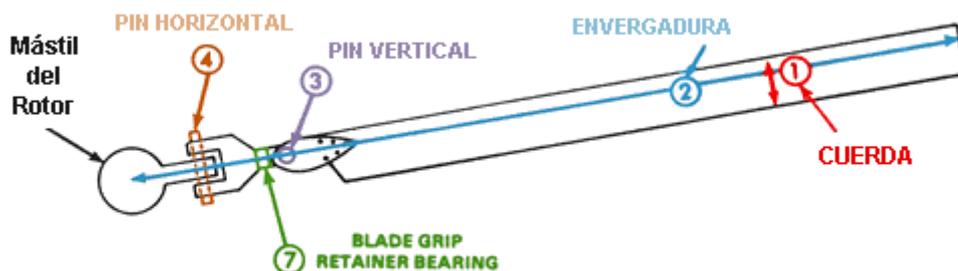
A reacción: Colocando toberas en los extremos de las palas. Provoca unas fuerzas centrífugas demasiado intensas.

Todos los anteriores sistemas tienen un rendimiento demasiado bajo, aunque la ventaja de no transmitir ninguna fuerza al fuselaje. La siguiente es la alternativa usada habitualmente.

En el eje: Las palas se mueven por un rotor principal en el fuselaje, aunque se provoca en este una fuerza de reacción contraria que ha de ser compensada.

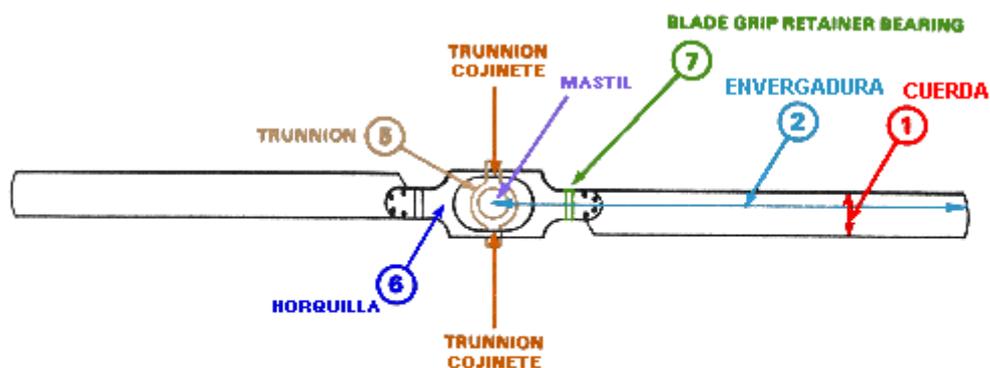
Rotores

El sistema mostrado en la siguiente figura corresponde a un sistema de rotor totalmente articulado:



ROTOR TOTALMENTE ARTICULADO

En el siguiente esquema se muestra un rotor semi-rígido, que no tiene ni pin horizontal ni pin vertical. En su lugar, el rotor se balancea por intermedio del cojinete que está conectado al mástil del rotor principal.



ROTOR SEMIRÍGIDO

Terminología básica con respecto a rotores

1.- CUERDA: Es la línea recta que une el borde de ataque con el borde de fuga. Es una dimensión característica del perfil.

2.- ENVERGADURA: Es la distancia de punta a punta del ala (o pala), independientemente de la forma que tenga.

3.- PIN VERTICAL (Vertical hinge pin): Es el eje de pivote que permite el adelantamiento (o retroceso) de la pala independientemente de las demás palas.

4.- PIN HORIZONTAL (Horizontal hinge pin): Es el eje que permite el pivote hacia arriba o hacia abajo de las palas (flapeo), independientemente de las demás palas.

5.- TRUNNION: Es el elemento que permite el "flapeo" de las palas.

6.- HORQUILLA (YOKE): Es el elemento estructural al cual las palas van fijadas y al cual van ligadas al mástil a través del trunnion y al cojinete del trunnion.

7.- BLADE GRIP RETAINER BEARING: Es el cojinete que permite la rotación de las palas sobre su eje longitudinal para permitir el cambio de paso.

8.- Torsión de la Pala (Blade Twist): Es una característica de construcción de las palas para que el ángulo de incidencia en la punta sea menor que en la raíz. Esta torsión de la pala ayuda a mantener la sustentación a lo largo de la misma incrementando el ángulo de incidencia en la raíz donde la velocidad es menor.

Compensación del momento de rotación

Mediante hélice lateral: Se monta lateralmente una hélice orientable, que puede ser usada para apoyar el avance del helicóptero, pero que resulta difícil de equilibrar para evitar variaciones en la dirección del helicóptero.

Mediante un chorro de cola

Mediante una hélice de cola con registro orientable: También contribuye al avance.

Mediante una hélice de cola de acción transversal: Es el método más popular

Helicópteros de varios rotores

En este caso se puede conseguir la compensación del momento de rotación haciendo girar los dos rotores en sentidos opuestos.

Puede haber rotores laterales, en tándem (uno tras otro), solapados y coaxiales.

Pero la complicación mecánica hace que estos modelos se usen menos que los basados en rotor de cola.

Bases de la producción de fuerzas de vuelo

El ascenso se produce gracias a la rotación a gran velocidad de las palas del rotor. Esto hace que el diseño de las mismas sea muy importante a la hora de la construcción del aparato. Así, en helicópteros de gran tamaño hay grandes variaciones, aunque en aeromodelismo suele utilizarse una pala de igual anchura y grosos en toda su longitud.

Esto varía en modelos de mayor tamaño, dado que el aire incide de diferente manera según recorremos la longitud de la pala, con lo cual se pueden diseñar palas más delgadas en el extremo, o más anchas en el centro para hacerse más delgadas en el extremo, etc.

En general, para helicópteros grandes, se habla de 0,2 CV/kg como la potencia necesaria para conseguir el vuelo del helicóptero, teniendo en cuenta todos los

factores (viento, resistencias...), incluido que no toda la potencia irá destinada al rotor. A esta potencia se la llama potencia ponderal.

La relación del helicóptero con su entorno tiene especial importancia. Así, cerca del suelo, en la elevación, se produce el efecto tierra, que proporciona un impulso extra al helicóptero, dado que el aire expulsado por las hélices rebota contra el suelo y proporciona más potencia al aparato. Asimismo, en el vuelo horizontal se aprovecha la presión adicional del aire para obtener más impulso.

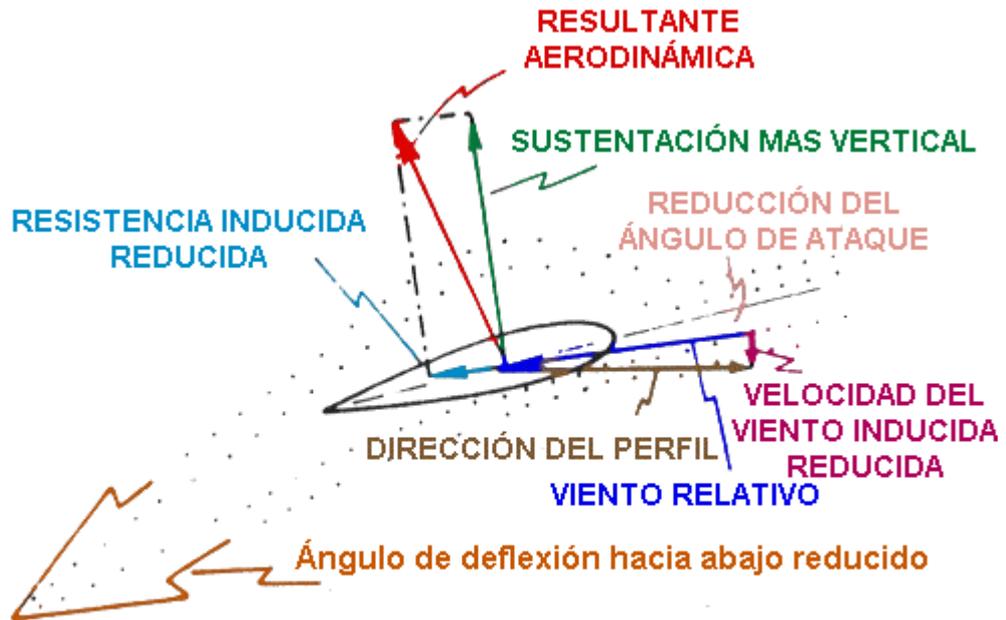
Efecto tierra

La alta potencia necesaria para mantener un vuelo estacionario, fuera de efecto tierra, se reduce cuando este vuelo es realizado en las proximidades de la tierra. El efecto tierra mejora las condiciones sobre un helicóptero. Tomamos para esto una proximidad con la tierra de la mitad del diámetro del rotor.

El incremento en las eficiencia de la pala, en su proximidad con el suelo, se debe a dos fenómenos diferentes.

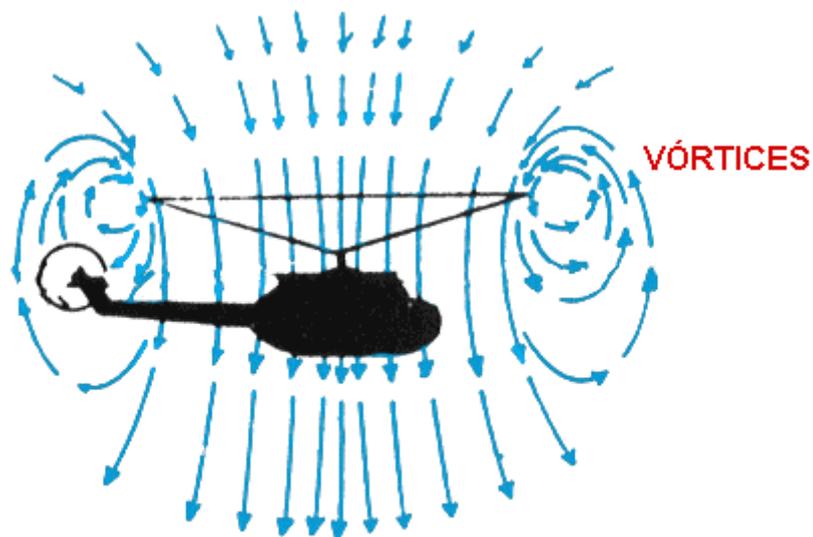
El primero y el más importante es la reducción de la velocidad del flujo de aire inducido. Puesto que el suelo interrumpe el flujo de aire bajo el helicóptero, se reduce la velocidad del flujo descendente inducido. El resultado es menos resistencia inducida y sustentación más vertical.

La sustentación necesaria para sostener un estacionario puede ser producida con un menor ángulo de ataque y menos potencia debido a que el vector de la sustentación se encuentra más vertical.

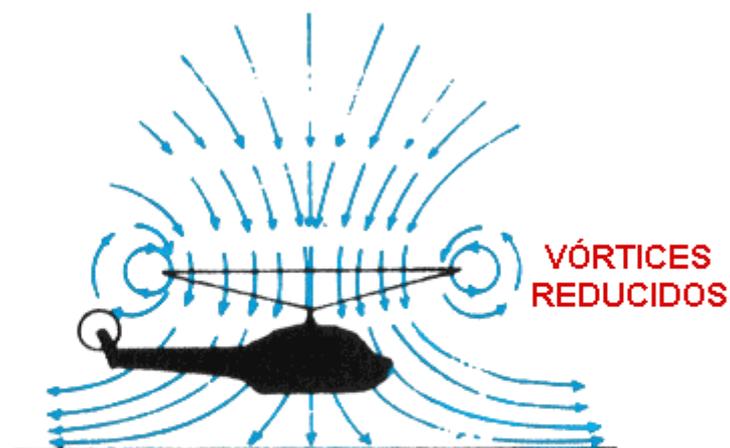


ESTACIONARIO CON EFECTO SUELO

El segundo fenómeno que sucede es la reducción de los vórtices en el extremo del rotor.



ESTACIONARIO FUERA DE EFECTO SUELO



ESTACIONARIO CON EFECTO SUELO

Cuando se está operando con efecto suelo, la parte externa y descendente del flujo de aire tiende a restringir la generación de los vórtices del extremo del rotor. Esto hace más efectiva la parte externa de las palas y reduce la turbulencia causada por la circulación de los remolinos.

La eficiencia del rotor en efecto suelo puede considerarse hasta una altura igual al diametro del rotor para la mayoría de los helicópteros.

Control del helicóptero

El control vertical del aparato se puede llevar a cabo de distintas maneras, siendo la más popular la de paso colectivo de la pala, que consiste en modificar el ángulo de incidencia de las palas de manera conjunta, siendo muy útil para el pilotaje en autorrotación o posición de planeo.

Otro sistema puede ser el control del número de revoluciones del rotor, con lo que se reduce la complejidad mecánica, aunque esta en desuso por no permitir precisamente la autorrotación.

El control de la dirección del fuselaje se realiza mediante ajuste con la compensación del rotor de cola, dificultado por el método de paso colectivo de la pala, pues este resta revoluciones al rotor de cola, que precisamente las necesita para modificar la dirección del aparato.

El control del desplazamiento horizontal del helicóptero es fundamental, y se realiza inclinando el rotor de tal manera que incida con el aire de modo que realice el avance deseado. Esto lo podemos hacer desplazando el centro de gravedad del helicóptero, el desplazamiento del eje del rotor, o mediante una cabeza basculante.

Todos estos sistemas dan problemas de lentitud o mecánicos, por lo que el más utilizado es el de paso cíclico de la pala. En lo que consiste este método es en que se puedan inclinar las palas del motor independientemente de la cabeza del rotor.

Controles físicos del helicóptero

Se debe utilizar una palanca para provocar la elevación o el descenso (que incluye un mando para determinar la potencia del motor), otra para la dirección del fuselaje (que comúnmente en helicópteros grandes corresponde a los pedales de giro), y otra para la dirección con respecto al eje vertical.

Esto nos hace pensar que con un pad doble, del tipo de los videojuegos, podría controlarse el aparato, pensando en una simulación por ordenador.

Con el primer control, llevamos a cabo el estrangulamiento del motor (regulación por número de revoluciones) o al paso colectivo de la pala.

El segundo acciona la inclinación de la pala en el rotor de cola. Un movimiento a la izquierda significa que el modelo girará hacia la izquierda (visto desde la posición del piloto), y lo mismo para la derecha.

El ultimo control acciona el disco oscilante y se controla el paso cíclico de la pala.

También, para un control más avanzado, se puede utilizar algún sistema más complejo, como puede ser una consola de control basada en potenciómetros, que nos permitiría variar todos los parámetros del vuelo del helicóptero.

Para ello deberíamos disponer de un driver específico para manejar la consola de instrumentos, que probablemente habría que desarrollar, lo que hace que en un principio desechemos esta idea.

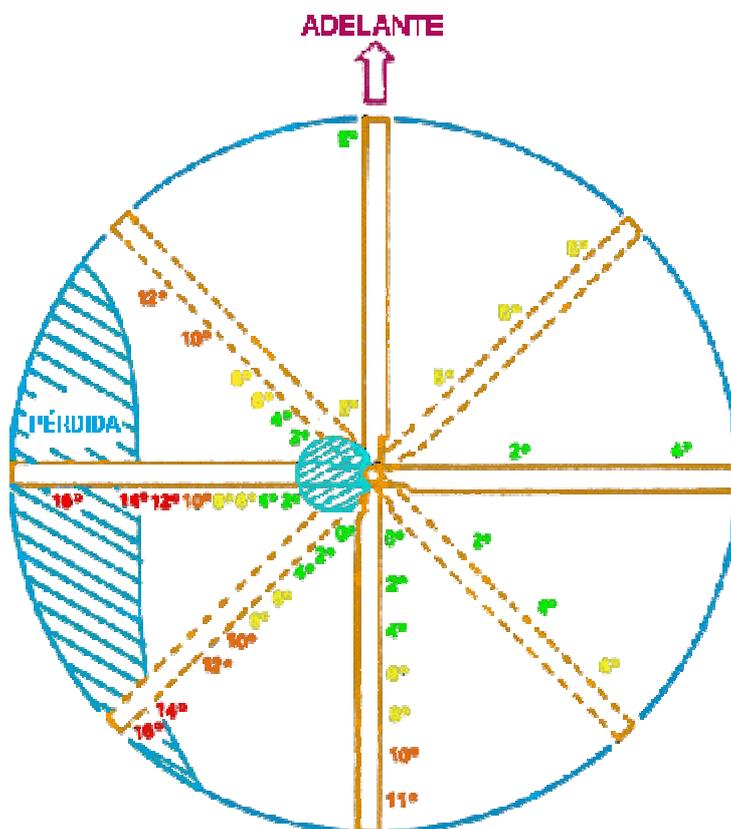
Suspensiones

Los diferentes tipos son:

- Articulada: Consiste en que las dos palas están conectadas por una articulación, con lo que las fuerzas de las dos palas se contraponen y se consigue un vuelo suave
- Con articulaciones individuales: Aquí las fuerzas se transmiten al cubo, con lo que se mejora la respuesta en el control
- Amortiguada: Como la primera opción, pero con un sistema de amortiguación que transmite parte de las fuerzas al fuselaje
- Rígida o no articulada: Esta provoca la reacción inmediata en el control de las fuerzas que intervienen

Presión asimétrica del aire en el vuelo hacia delante

Durante el vuelo en horizontal de un helicóptero se produce un efecto que produce la inclinación transversal del modelo y puede llegar al vuelco de este. Esto es debido a que la pala que se mueve contra la presión del aire en el giro del rotor es impulsada con una fuerza mayor que la pala que se mueve con la presión del aire.



Esto afecta en mayor medida a rotores que giran más despacio, dado que entonces la relación con la velocidad propia es mayor, por eso se tiende siempre a incrementar el número de revoluciones del rotor.

Articulaciones de batida y oscilación

La articulación de batida se utiliza para disminuir el efecto que tiene en las palas la presión asimétrica del aire, esto es, el hecho de que una pala esté acelerada y la otra retardada.

La articulación de oscilación se utiliza para que la pala pueda girar en la dirección del rotor, y así evitar que la pala se adelante a su posición primitiva, dado que tiende a girar cada vez más deprisa (por el desplazamiento del centro de gravedad de esta con la velocidad). Estas articulaciones aún no se usan en aeromodelismo.

Autorrotación

Se denomina autorrotación al vuelo planeado de un helicóptero sin propulsión, debido a que las palas continúan girando como un molinete y producen un impulso de ascenso.

La manera de intentar mantener el número de revoluciones del motor es retirar el ángulo de incidencia que estaba dispuesto para el vuelo normal, mediante la palanca de inclinación de pala.

Es preciso para esto que en el diseño del aparato se incluya un piñón libre en la propulsión del rotor principal, que hará que el rotor de cola se pare y que el control de dirección se realice exclusivamente con el rotor principal. Podría hacerse que el rotor de cola no se parara y tomara energía del movimiento del rotor principal, pero en este caso se le restaría a este, y puede que no tuviera suficiente potencia para enderezar el helicóptero y reposarlo, lo cual hace que normalmente no se utilice esta variante.

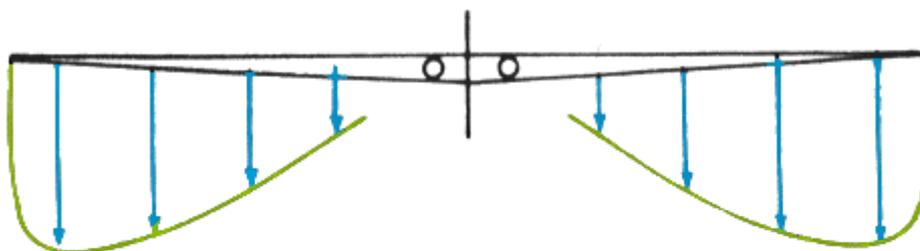
Fases de vuelo peligrosas

Existen fases del vuelo en las que el paso a la autorrotación no es posible.

Dos zonas de este tipo son el vuelo a baja altura con velocidad alta, el vuelo en suspensión estacionario sin velocidad de avance.

Otra zona peligrosa es la zona del propio chorro del rotor del helicóptero. Si se inicia un descenso vertical muy rápido en esta zona, el helicóptero caerá en este chorro y se disminuye el impulso de ascenso. La única manera de evitarlo es *empujar* el helicóptero hacia delante y salir del chorro.

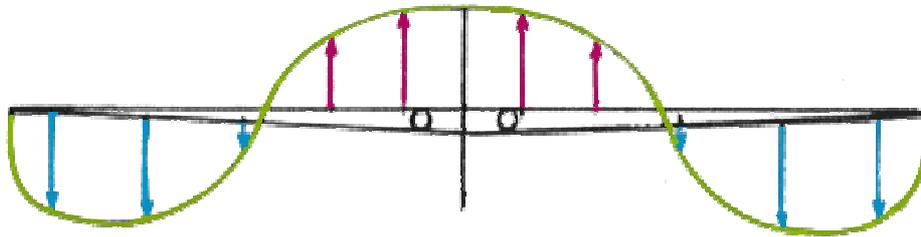
Este dibujo muestra el flujo de aire a lo largo de la pala, en vuelo estacionario normal:



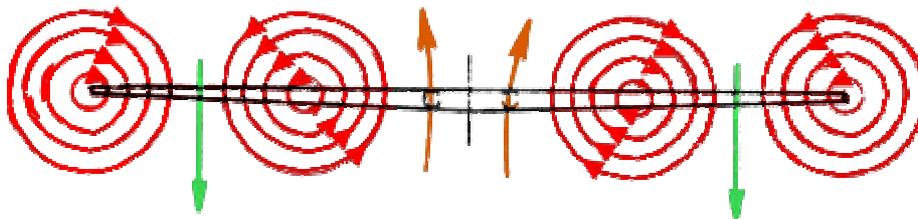
VELOCIDAD DEL FLUJO INDUCIDO DURANTE EL ESTACIONARIO.

La velocidad descendente es mayor en la punta de las palas donde la velocidad por rotación es mayor. El siguiente dibujo nos muestra el patrón del flujo de aire

inducido, a lo largo de la pala, durante un descenso que conduce a un asentamiento con potencia.



El descenso es tan rápido que el flujo de aire inducido de la porción inferior de la pala es ascendente más que descendente. El flujo de aire hacia arriba, causado por el descenso, ha vencido al flujo de aire hacia abajo de las palas. Si el helicóptero desciende en estas condiciones, con insuficiente potencia para disminuir o parar el rango de descenso, entrará dentro del *anillo turbillonario*, que se puede ver a continuación.



Estabilidad

Un helicóptero no puede llegar a ser un sistema estable, dado que nunca volverá a su posición inicial después de un cambio de estado. Por eso debemos tratar de hacerlo indiferente, mejor que inestable.

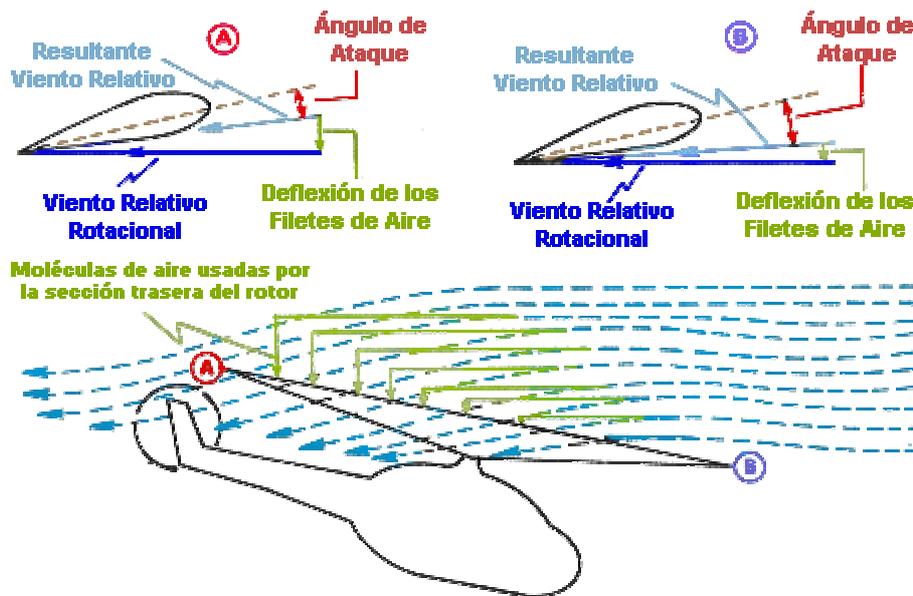
Para ello hay tres métodos principales:

- Método Bell: Aquí se instalan unas varillas de estabilización, para que si se inclina el aparato, las varillas permanecen en su posición y colaboran a que las palas del rotor lo hagan de igual forma.
- Método Hiller: Este consiste en tener un rotor auxiliar unido al principal mediante varillas, que pueden trabajar como simples pesos de estabilización (cuando no se controlan), o cambiando su posición en el espacio si se controlan mediante el disco de paso cíclico.
- Rotor Schlüter: Este es usado principalmente en aeromodelismo

También se ha usado a veces una combinación del método de Bell y Hiller, en que se convierten los pesos de la vara de estabilización de Bell en alas tipo Hiller, suprimiendo los amortiguadores Bell.

El vuelo transversal

En vuelo hacia adelante, el aire que pasa a través de la parte posterior del disco del rotor tiene un flujo de aire descendente (*downwash*) mayor que la parte delantera.



EFECTOS DEL FLUJO TRANSVERSAL

El flujo de aire descendente en la parte trasera del disco provoca un reducido ángulo de ataque, resultando en una menor sustentación. Debido a que el flujo de aire es más horizontal, un mayor ángulo de ataque y una mayor sustentación se obtienen en la parte frontal del disco. Esta diferencia entre la parte trasera del disco y la parte frontal es llamada **flujo transversal**. Este **flujo transversal** causa diferencias de resistencia entre ambas partes del disco, resultando en vibraciones que son fácilmente reconocidas por el piloto.

Dinámica de las palas del rotor

Las fuerzas que actúan sobre la pala del rotor pueden ser estáticas (cuando está parado), dinámicas (cuando está en movimiento) o aerodinámicas (que son las resultantes de la interacción con el aire).

Las fuerzas estáticas se producen en estado de parada del rotor y se pueden controlar con facilidad. Aquí entraría el efecto que el peso de las palas tiene sobre el helicóptero cuando estas están en reposo.

Las fuerzas dinámicas solo surgen durante la rotación del sistema del rotor, y sus componentes esenciales pueden determinarse en la comprobación de las fuerzas estáticas. Aquí entraría el efecto que el peso de las palas tiene sobre el helicóptero cuando estas están en movimiento.

La fuerza centrífuga y los efectos de la sustentación pueden ser mejor entendidos con un gráfico. Primero vemos un eje de rotor y una pala rotando.



FUERZA CENTRÍFUGA

Este será el mismo rotor cuando una fuerza vertical le es aplicada en la puntera de la pala.



SUSTENTACIÓN Y FUERZA CENTRÍFUGA

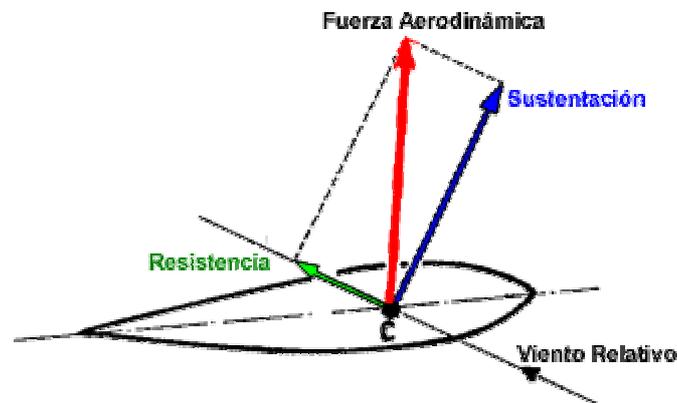
La fuerza aplicada es la sustentación producida cuando las palas aumentan su ángulo de ataque. La fuerza horizontal es la fuerza centrífuga generada por el rotor al girar. Debido a que la raíz de la pala está sujeta al árbol, sólo el otro extremo tiene la libertad de moverse y se obtiene una resultante en la pala como muestra la siguiente figura.



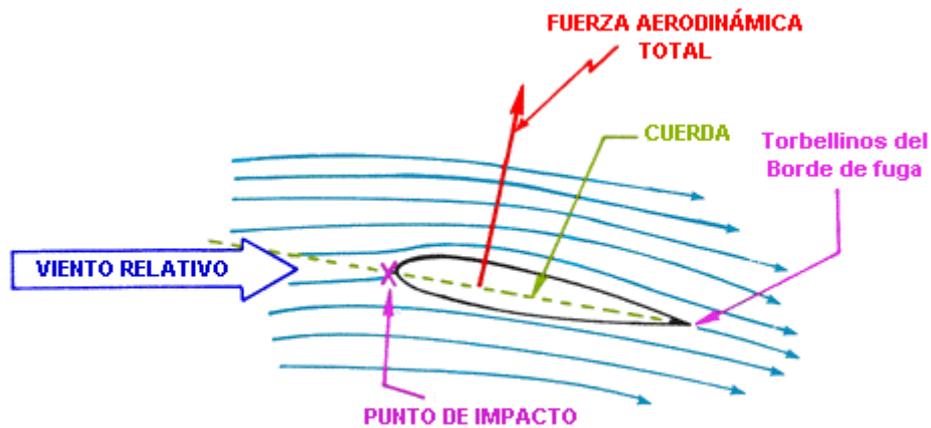
RESULTANTE DE SUSTENTACIÓN Y FUERZA CENTRÍFUGA

La posición de la pala es la resultante de dos fuerzas: la sustentación y la fuerza centrífuga.

Las fuerzas aerodinámicas aparecen por la interacción entre el sistema del rotor y el aire sobre las palas de este. Aquí entra, por ejemplo, el efecto que tiene el ángulo de incidencia de las palas del rotor con el aire, que provocará un ascenso/descenso más o menos intenso. Así, estas fuerzas son las que provocarán que las palas formen un ángulo cónico, ya que si las fuerzas centrífugas tratan de estabilizar las palas en una posición horizontal, las fuerzas aerodinámicas las empujan hacia arriba.



DESCOMPOSICIÓN DE LA FUERZA AERODINÁMICA



FLUJO SOBRE UN PLANO AERODINÁMICO

Torsión de la pala, y su influencia sobre el pilotaje

El efecto torsión se produce por el hecho de que el centro de gravedad, sobre el que actúa la fuerza centrífuga, y el punto medio de impulso de ascenso, sobre el que actúan las fuerzas aerodinámicas, normalmente no suele coincidir.

Lo ideal sería que coincidieran. Lo que suele hacerse es intentar colocar el punto medio de impulso por detrás del centro de gravedad. Esto se consigue colocando mayor peso en la punta de la pala, y en general en la parte delantera del helicóptero.

Con esto, se conseguirá que la distancia entre ambos puntos no sea muy grande y que por tanto las fuerzas de torsión tampoco lo sean, con lo que con unas palas de rotor suficientemente rígidas se podrá controlar el aparato.

Otro problema que se puede encontrar es el debido a un montaje asimétrico de las palas del rotor. Si esto sucede, la línea que une los centros de gravedad de ambas palas estará separada del eje de rotación, lo que da lugar a un fuerte desequilibrio, con lo cual se debe evitar.

La solución que se suele utilizar es la de emplear cabezas de rotor equipadas con conexiones de la pala en forma de horquilla con un único tornillo de fijación. Así, la pala puede oscilar y colocarse en la línea de acción del centro de gravedad de la pala.

4. IMPLEMENTACIÓN GRÁFICA

En esta sección vamos a ir describiendo los diferentes aspectos que forman parte del aspecto gráfico del simulador.

DIRECTX

La Api 3D que hemos empleado en el proyecto es la versión 8.0 de DirectX. Esta Api es propietaria de Microsoft (www.microsoft.com). Otra de las opciones posibles era usar OpenGL (www.opengl.org). Realmente los autores de este proyecto han trabajado con ambas APIs. Y aunque OpenGL sea una API mucho más sencilla y directa, hay que reconocer que con DirectX se tiene más control. DirectX v8.0 incorporaba los conceptos de *PixelShader* y *VertexShader*, que estábamos muy interesados en probar. Finalmente, y por falta de tiempo, no se usaron.

Es imposible en el poco espacio que tenemos aquí, explicar con profundidad todos los aspectos de esta API. Sin duda alguna lo mejor es bajarse el SDK, disponible en <http://www.microsoft.com/directx/>. En el SDK se incluyen numerosos ejemplos y una amplia documentación.

Una vez conocida a fondo la API, el siguiente paso es empezar a pensar en optimizaciones posibles de código. Hoy en día, las tarjetas aceleradoras poseen una capacidad de potencia igualable a la CPU y en muchos casos superior. El mejor modo de optimizar un engine gráfico es intentar paralelizar ambos procesadores (la CPU y la GPU). Para maximizar esa paralelización es importante realizar el menor número de llamadas a la API (cuantas más llamadas, más dependencia crearemos entre ambos procesadores). Uno de los modos de alcanzarlo es emplear tiras de triángulos (strips). De modo que con una sola llamada podemos pintar un buen número de triángulos. Mientras se están pintando esos triángulos la CPU queda libre para hacer cálculos en paralelo.

Llegar a comprender las posibles optimizaciones de código para sacar máximo rendimiento de la tarjeta es algo que requiere bastante práctica y conocimiento. En la página de Nvidia (www.nvidia.com) existen numerosos papers que detallan todo tipo de reglas a seguir.

Todo el control de la API DirectX 8.0, queda encapsulado en las siguientes clases que pasamos a describir.

D3Dwrapper

La misión de esta clase es encapsular la API Direct8, de modo que simplifica muchas de las operaciones a realizar. Por ejemplo, la selección de un modo de video, la carga de texturas...

D3Dfont

Esta clase es la encargada de pintar texto en pantalla. Es la que usa internamente D3Dwrapper para mostrar información. Pero igualmente puede ser usada fuera de la clase D3Dwrapper.

StateManager

Las tarjetas gráficas funcionan como una máquina de estados. De modo que a la hora de pintar las primitivas el resultado final depende del estado actual de la tarjeta. Algún ejemplo de estados son los siguientes:

- Modo de relleno (sólido, alámbrico)
- Iluminación activada / desactivada
- Tipo de sombreado
- Tipo de textura
- Tipo de transparencia empleada

La función de StateManager es filtrar los cambios de estados para que se produzca el menor número de llamadas a la API. De modo que si por ejemplo, activamos el modo Alámbrico y este modo ya está seleccionado entonces, StateManager no vuelve a llamar a la API, ahorrando una llamada.

DynamicVBManager

Los VertexBuffers son los buffers donde se guarda la información de los vértices. Esta información de los vértices es la que luego se usa para pintar las primitivas en pantalla. La función de DynamicVBManager es actuar como un gestor de memoria de buffers de vértices.

Shader, ShaderPass

Estas clases son las encargadas de gestionar los cambios de estados pero a un nivel más alto que la API. Un Shader representa un determinado aspecto (Por ejemplo, se puede crear el Shader “oro”, que intenta imitar las propiedades del oro). Un Shader puede necesitar de varios repintados (es decir, que la tarjeta pinte el triángulo varias veces acumulándolo en el buffer), y para ello emplea los ShaderPass.

Todo Shader debe ser validado antes de ser empleado. Validar un Shader implica llamar a la función Validate, cuya misión es comprobar que dicho Shader es compatible con la tarjeta gráfica actual.

TRANSFORMACIONES GEOMÉTRICAS

1. Código relacionado

Matriz.cpp
Matriz.h

2. Descripción

Una matriz $M: \mathbb{R}^3 \rightarrow \mathbb{R}^3$ se denomina una transformación lineal ya que nos lleva de vectores en 3 dimensiones a vectores en 3 dimensiones. La matriz cero es aquella matriz con todas sus entradas a 0. La matriz identidad es aquella matriz (denotada I) con 1 en su diagonal y 0 en el resto de las entradas.

Con este tipo de matrices podemos representar transformaciones muy útiles a objetos 3d como la rotación y el escalado. Pero una matriz 3x3 es insuficiente para incorporar al mismo nivel de representación la operación translación. Para ello es necesario introducir el concepto de transformación homogéneas. Un punto tridimensional se representa en coordenadas homogéneas con una cuarta dimensión igual 1, mientras que un vector tridimensional se representa con una cuarta dimensión igual a 0.

$$\begin{aligned} [x, y, z] &\Rightarrow [x, y, z, 1] && \text{Esta clase se implementa en } \textit{Point3.h} \\ [x, y, z] &\Rightarrow [x, y, z, 0] && \text{Esta clase se implementa en } \textit{Vector3.h} \end{aligned}$$

El proceso de transformar un vector 4D por una matriz es el siguiente

$$[x', y', z', w'] = [x, y, z, w] \begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{bmatrix}$$

A continuación describimos las matrices equivalentes para las operaciones más comunes.

Translación: $T(d_x, d_y, d_z)$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ d_x & d_y & d_z & 1 \end{pmatrix}$$

Rotación $R_x(\phi)$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \phi & -\sin \phi & 0 \\ 0 & \sin \phi & \cos \phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotación $R_Y(\phi)$

$$\begin{pmatrix} \cos \phi & 0 & \sin \phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \phi & 0 & \cos \phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotación $R_Z(\phi)$

$$\begin{pmatrix} \cos \phi & -\sin \phi & 0 & 0 \\ \sin \phi & \cos \phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Escalado $T(s_x, s_y, s_z)$

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Al trabajar con coordenadas homogéneas vamos a usar matrices 4x4. Normalmente la

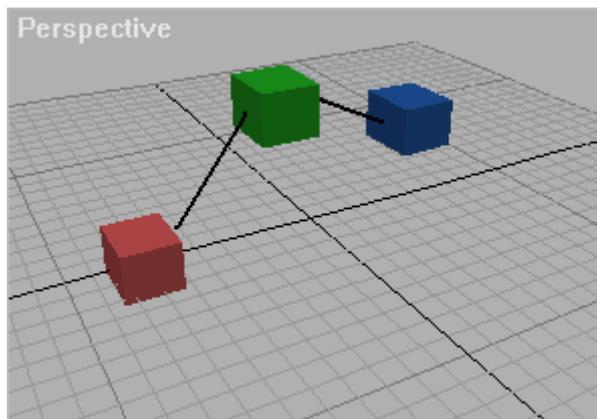
columna de la derecha es $\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$ por lo que en nuestra implementación de la clase Matriz

vamos realmente a trabajar con una matriz 4x3. En algunos casos esto no será válido (como por ejemplo la matriz de proyección empleada por la cámara) y tendremos que emplear código especializado en el caso más general.

Con el fin de optimizar las operaciones más frecuentes, en la clase Matriz se incluyen métodos para trasladar, rotar y escalar tanto en coordenadas locales (los que llevan el prefijo Pre y que equivale a multiplicar por la izquierda) como en coordenadas de mundo (los que no llevan prefijo y que equivale a multiplicar por la derecha)

Matrix
<pre> +IdentityMatrix() +PreScale(in x : float, in y : float, in z : float) +PreTranslation(in x : float, in y : float, in z : float) +PreRotX(in angle : float) +PreRotY(in angle : float) +PreRotZ(in angle : float) +Scale(in x : float, in y : float, in z : float) +Translation(in x : float, in y : float, in z : float) +RotX(in angle : float) +RotY(in angle : float) +RotZ(in angle : float) </pre>

Cada objeto presenta en nuestra escena (ver el apartado del grafo de escena) tendrá una matriz que lo situara en coordenadas con respecto a su sistema de coordenadas superior. En un sistema sin jerarquía los objetos expresan en su matriz su orientación y posición con respecto a las coordenadas de mundo. Si un objeto es hijo de otro, su matriz expresa la orientación y posición con respecto al sistema de coordenadas del padre. Veamos un ejemplo:



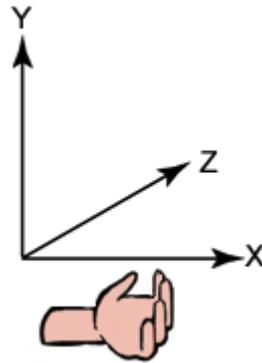
En esta escena de muestra cada uno de los objetos posee una matriz de orientación del siguiente modo:

$$M_{objeto} = M_{translacion} M_{rotacion} M_{escalado}$$

La caja roja es padre de la verde y a su vez la caja verde es padre de la azul. Para obtener las coordenadas de mundo de la caja azul debemos formar la siguiente matriz:

$$M = M_{azul} M_{verde} M_{rojo}$$

Deberíamos añadir luego la matriz de camera y de proyección si quisiéramos coordenadas en view. Es importante destacar el sistema de coordenadas que estamos empleado en nuestro sistema y que se suele denominar de *mano izquierda*. En el dibujo se muestra claramente el convenio a seguir.



Antes de pintar cada objeto es necesario fijar en el rasterizador la matriz que convertirá las coordenadas locales a mundo. En nuestro caso estamos empleado la API DirectX a través de una clase Wrapper que internamente traduce nuestra clase *Matriz* a *D3DXMatrix* que es la empleada por DirectX.

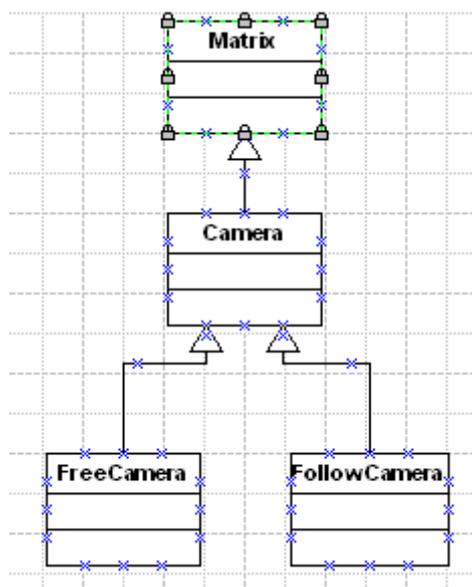
SISTEMA DE CAMARAS

1. Código relacionado

Camera.cpp
Camera.h
FreeCamera.cpp
FreeCamera.h
FollowCamera.cpp
FollowCamera.h

2. Descripción

El sistema actual de cámaras que estamos empleando para el proyecto se jerarquiza el siguiente grafo.

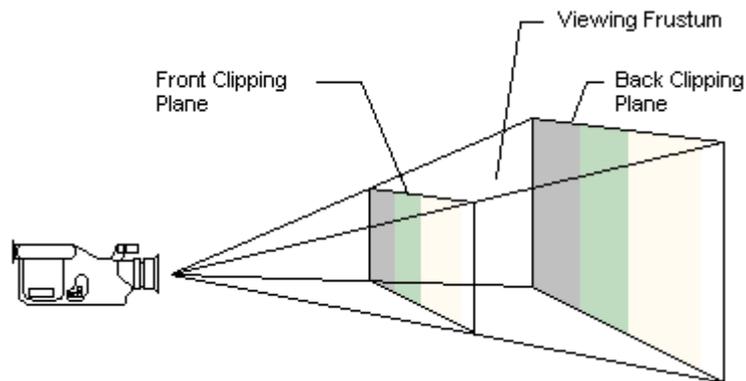


A continuación pasamos a describir brevemente el funcionamiento de cada una de las clases. Para una descripción profunda de cada uno de los métodos es necesario consultar la documentación que existe sobre el conjunto de clases.

1. Camera

En esta clase se añade toda la funcionalidad básica para cualquier cámara. Una cámara, además de ser una matriz de orientación (de ahí la herencia de la clase Matriz) posee una matriz de proyección *cónica* (no vamos a tener en cuenta las proyecciones *ortogonales*) generada a partir de los siguiente parámetros:

- FOV: es el ángulo de apertura de la cámara
- NearPlane y FarPlane son los planos cercano y lejano de la pirámide de visualización.



2. FreeCamera

Una cámara free es aquella que puede moverse libremente por el entorno tridimensional. Se emplean como dispositivos de entrada el ratón y el teclado para permitir girar la cámara en 2 grados de libertad y avanzar o retroceder en la dirección apuntada por el eje Z de la cámara.

3. FollowCamera

Una cámara follow es aquella que está ligada a un nodo del grafo de escena (ver documentación sobre el grafo de escena) y le sigue según este se mueve. Se emplea un sistema de muelles para hacer que el movimiento de la cámara sea lo más suave posible. Los muelles van puestos en la posición de la cámara y en el punto de interés de la cámara. Para la integración se emplea la siguiente fórmula:

$$F = ma = -k_s x - k_d v$$

Donde, x es la distancia al muelle, k_s es la constante de muelle de Hook, k_d es la constante de disipación y v es la velocidad a la que se está moviendo el punto de fijación del muelle.

GRAFO DE ESCENA

1. Código relacionado

Mesh.h
Mesh.cpp
MeshNode.h
MeshNode.cpp
Node.h
Node.cpp
SceneGraphManager.cpp
SceneGraph.h

2. Descripción

Todo la geometría que se visualiza en el mundo debe formar parte del grafo de escena. El grafo de escena es un grafo acíclico dirigido (DAG), donde cada nodo representa un objeto gráfico y la jerarquía la determina la posición de los nodos. De modo que un determinado nodo expresa su sistema de coordenadas con respecto al sistema de coordenadas del padre.

Todo objeto que quiera integrarse en el grafo de escena debe heredar de la clase Node e implementar aquellos métodos que considere necesarios. La clase Node tiene todos los métodos implementados por defecto. A continuación describimos cada uno de los métodos que forman parte de la clase Node.

void SetName(const char *name);
const char *GetName(void) const;

Estos dos métodos son para fijar y obtener el nombre del nodo respectivamente. Al heredar de esta clase no es necesario reimplementar estos métodos.

void SetParent(Node *node);
Node *GetParent(void) const;

Estos dos métodos son para fijar y obtener el padre del nodo.

int GetNumChildren(void) const;

Devuelve el número de hijos del Nodo.

void AddChild(Node *child);

Añade un hijo al nodo.

Node *GetChild(int i) const;

Obtiene el hijo *i*-ésimo del nodo.

void SetMatrix(const Matrix &matrix);
Matrix GetMatrix(void) const;

Métodos para determinar la matriz del nodo. La matriz de un nodo representa la posición y la orientación del nodo con respecto al padre.

virtual bool ApplyCulling(void);

Indica si en este nodo se debe aplicar Frustum Culling. El Frustum Culling elimina objetos cuyo caja de envoltura no entre en la pirámide de visualización de la cámara.

void PropagateHierarchicalBoundingBox(void);
void GetHierarchicalBBox(AABBox &bBox);

Estos métodos se emplean para formar una jerarquía de cajas que envuelvan a al grafo de escena. Gracias a esta estructura, se puede determinar muy rápidamente que nodos son visibles a partir de la posición de la cámara.

virtual bool GetLocalBBox(AABBox &bBox);

Devuelve la caja de envoltura del nodo.

virtual bool IntersectRay(const Point3 &org, const Point3 &dir);

Determina si un rayo lanzado desde la posición *org* y con dirección *dir*, interseca al nodo.

virtual bool Update(float time);

Actualiza el nodo en el tiempo. Esta función se implementa en aquellos objetos cuya representación dependa del tiempo (por ejemplo objetos animados).

virtual bool Draw(D3DWrapper *raster, Camera *camera = 0);

Esta es la función a partir de la cual el objeto se pinta en pantalla.

Uno de los principales nodos empleados es el *MeshNode*. La clase *MeshNode* representa un nodo que contiene una malla triangular en su interior. A su vez *MeshNode* contiene en su interior una clase *Mesh* que es la que emplea para representarse en pantalla.

Para la lectura de mallas hemos empleado el formato *.X* de DirectX. Junto con el SDK de DirectX viene un plugin de exportación para 3DstudioMax 3.0 que es que hemos empleado para exportar la jerarquía de mallas que forman parte del helicóptero. Esto, que inicialmente fue hecho para visualizar rápidamente mallas, es la solución final que hemos adoptado. Sin duda, no es la más elegante, debido a que el

format .X es demasiado genérico. En futuras versiones del simulador incorporaremos un formato propio de malla, así como el correspondiente plugin de exportación para el 3dStudioMax.

La última clase en juego es *SceneGraphManager*. Esta es la clase encargada de gestionar todo el grafo de escena. Una vez creado el grafo de escena, con esta clase podremos pintarlo en pantalla y chuequearlo con respecto a colisiones de rayos. El *SceneGraphManager* aprovecha la estructura jerárquica del grafo de escena para optimizar todos los cálculos. Con esa misión construye una serie de cajas jerárquicas (para cada nodo crea una caja que envuelve a dicho nodo y a todos sus hijos) que optimizan tanto las funciones de pintado como las funciones de trazado de rayos.

REPRESENTACIÓN DEL CIELO

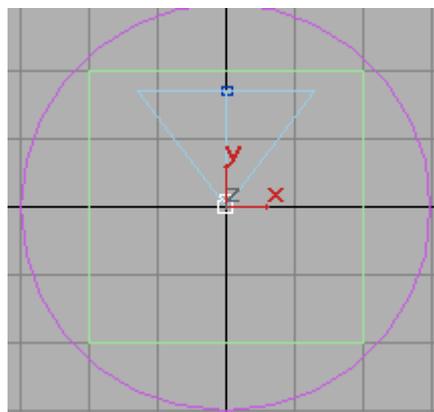
1. Código relacionado

SkyBox.cpp
SkyBox.h

2. Descripción

La técnica SkyBox nos va a permitir aumentar la complejidad visual de la escena sin apenas carga para la tarjeta gráfica o la aceleradora. Es ideal cuando queremos renderizar fondos que están muy lejanos con respecto a la posición de la cámara. De hecho, como su nombre indica se suele emplear para el pintado del cielo.

Básicamente, un SkyBox es un cubo con cada una de sus 6 caras texturadas. Las texturas de cada una de las caras son el resultado de render off-lines con una cámara de 90 grados de apertura. En estos renders usamos geometría lo más complicada posible pues es un cálculo que solo se hace off-line y que por tanto no afecta al rendimiento de tiempo real..

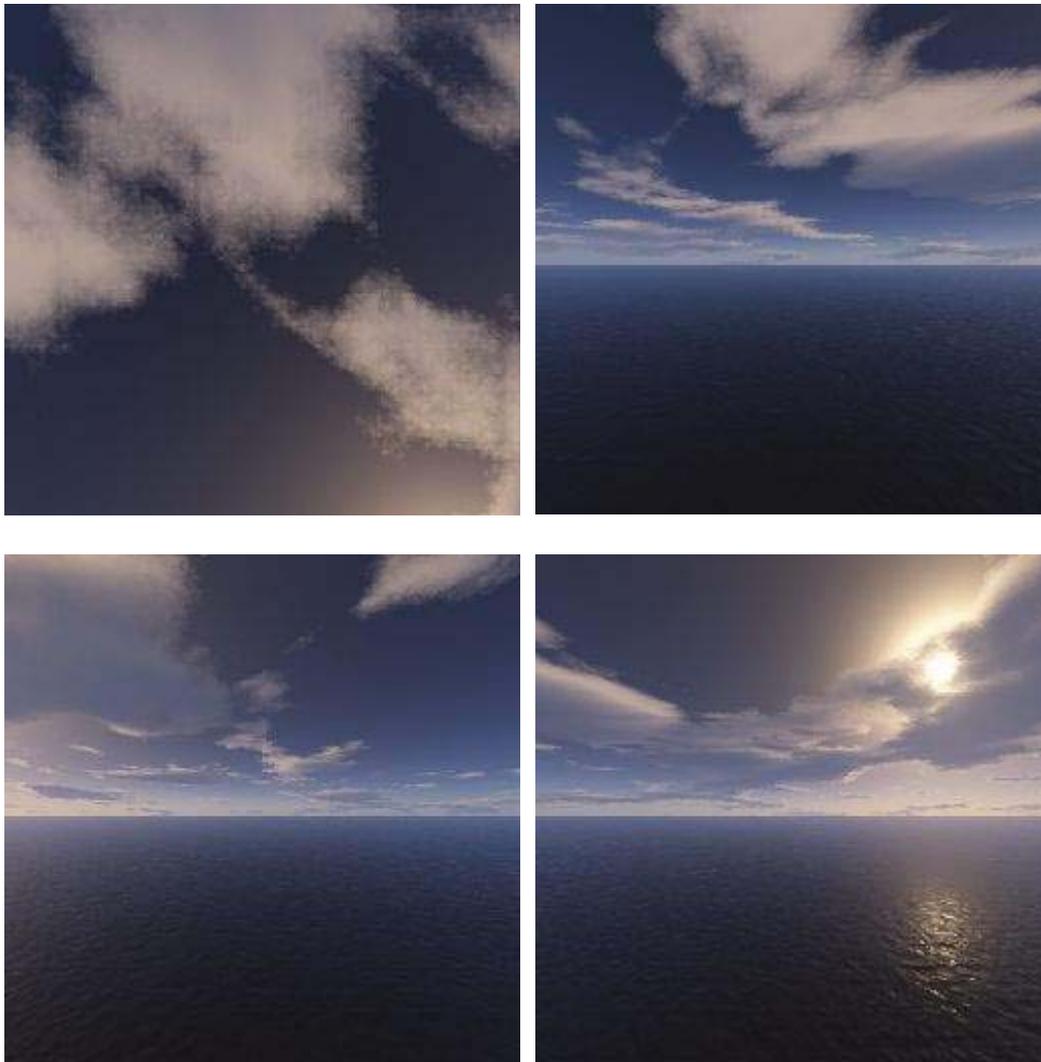


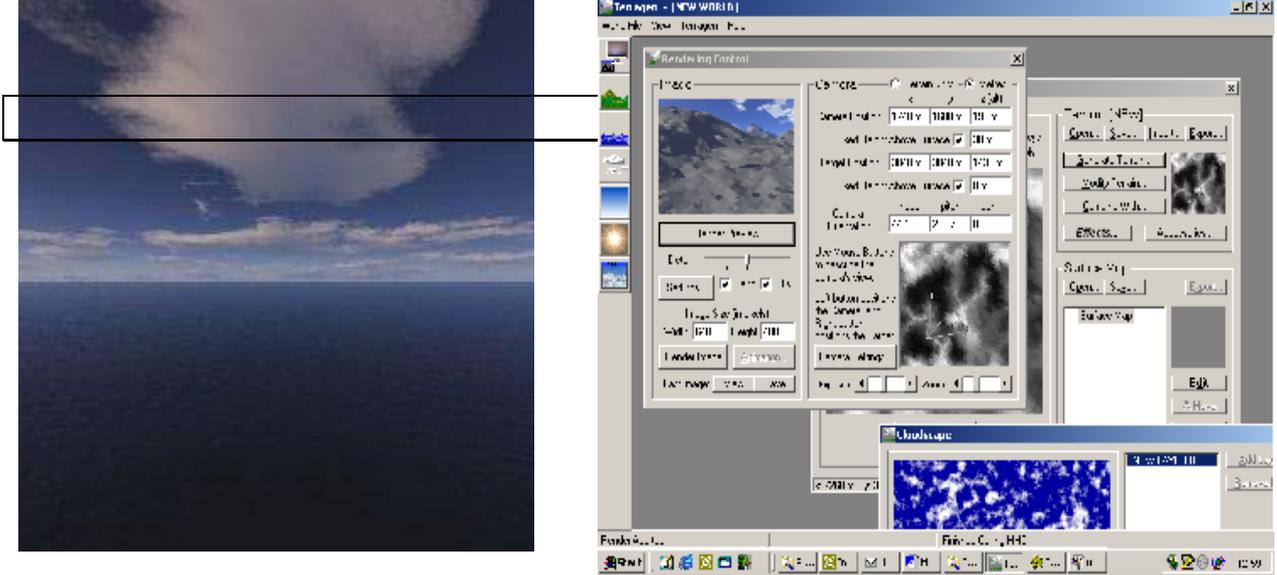
Una cámara centrada en el origen captura una de las caras del SkyBox

Una de las suposiciones importantes que hemos tomado en nuestro trabajo es que el SkyBox esta a una distancia infinita relativa a la posición de la cámara. Esto, en términos de implementación quiere decir que nuestros movimientos en la escena no afectan a la orientación del SkyBox.

En el cálculo de los SkyBoxes hemos empleado un programa llamado Terragen que esta especializado en el render de terrenos. El motor de generación del cielos de Terragen es impresionante. La ultima versión disponible de este programa se puede encontrar en www.planteside.co.uk. Con este programa, generamos un terreno y un cielo procedualmente y tomanos 5 capturas (la tapa de abajo del cubo no nos interesa ya que nunca se ve realmente) con una cámara centrada en el origen y con 90 grados de apertura (zoom 1.0 en Terragen).

A continuación mostramos las 6 caras de uno de uno de los skyboxes empleados para el simulador. Estas texturas, correctamente montadas en el skybox ocultan la geometría del cubo y dan la sensación de estar ante un fondo esferizado.





Si bien esta es una técnica bastante empleada en numerosos videojuegos, no es la única. El principal inconveniente que tienen los SkyBox es que es muy complicado luego animarlos en tiempo real. Esto sería interesante por ejemplo para animar varias capas de nubes. Para ello, se suelen usar lo que se denomina con el término inglés *dome*, que es una semiesfera poligonal. Montando varias semiesferas poligonales de estas y animándolas individualmente se consigue el efecto deseado. Por supuesto se pueden combinar ambas técnicas y conseguir resultados espectaculares.

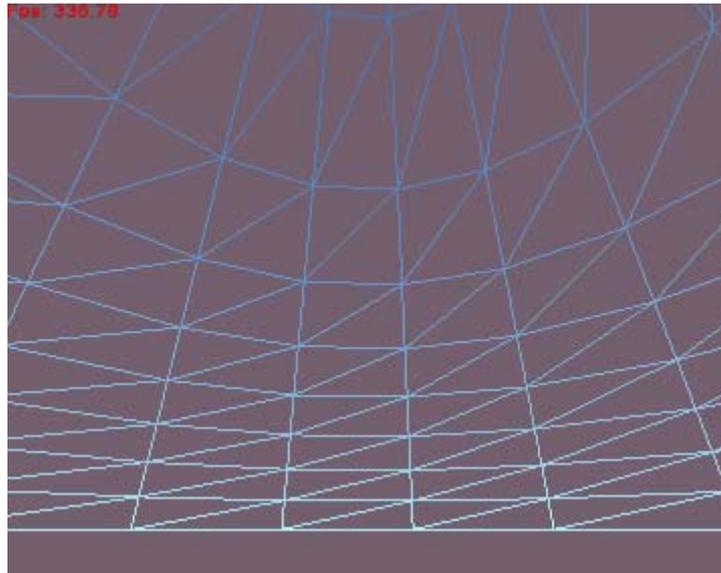
Representación mediante semiesferas.

Tras varias pruebas, decidimos mejorar la técnica de representación del cielo. El cielo ocupa gran parte del rango de visión, por lo que su correcta representación aportara bastante realismo al total de la simulación. El método anteriormente descrito sufre del principal problema de que el fondo tiene que ser estático.

Los requisitos que deberían cumplir un buen sistema de representación de cielos son los siguientes:

- Debe permitir animación de una o varias capas de nubes
- Debe poder representar cualquier hora del día

Para ello, e inspirados en [25], mediante el método de la semiesfera, el cielo se representa en varias pasadas. Todas sobre la misma geometría, que se muestra en la siguiente imagen.



En la primera pasada renderizamos la iluminación general de la atmósfera. Esta iluminación depende de la posición del sol. Hemos dejado para futuras versiones la posibilidad de generar un cielo a cualquier hora del día. De momento, los colores que se aplican en los vértices de la semiesfera son siempre los mismos. Dichos colores son interpolados y obtenemos una siguiente capa que se muestra en la siguiente imagen.



El siguiente paso consiste en añadir las capas de nubes. Dichas capas además pueden animarse unas con respecto a las otras. En la primera entrega del proyecto hemos obtenido buenos resultados con una única capa de nubes animadas ligeramente. Se podría conseguir más realismo haciendo que la velocidad de las nubes dependiera de algún factor. En nuestro caso, las nubes siempre se desplazan a la misma velocidad. Estas capas de nubes van mezcladas con la anterior capa de atmósfera, para dar un resultado final que se muestra en la siguiente imagen.

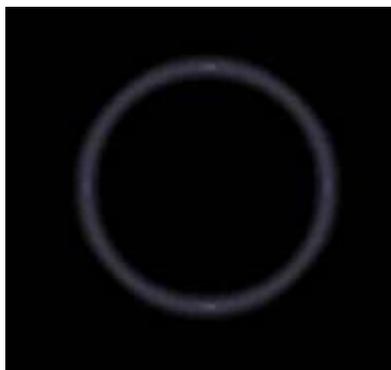


SISTEMA DE FLARES

Aprovechando que el diseño del grafo de escena permite comprobar si un determinado rayo colisiona con algún objeto, hemos incorporado al engine gráfico la capacidad de representar *Flares*.

En el ámbito de la programación 3D, se denominan *Flare (Lens Flare)* al conjunto de coronas o halos circulares que salen al impactar el sol directamente sobre la cámara. El halo aparece porque el material de la lente de la cámara refracta la luz en diferentes ángulos según el espectro de frecuencia.

En la práctica asociamos este efecto con el brillo, y para su representación se suele ocurrir al pintado de sprites texturados en pantalla. Nosotros hemos empleado la textura que mostramos a continuación (aunque a veces se suele usar más de una textura, generalmente no conviene abusar de este efecto, porque puede quedar muy artificial).



Esta textura se pinta aditivamente, para se sume al contenido de lo que hay ya pintado. La posición de la textura depende del centro de la cámara y de la proyección de la posición del sol. Con esos dos puntos se lanza un rayo y donde colisione con el plano de representación de la cámara es pinta la textura. Para incrementar más el realismo del algoritmo, hemos recurrido a algunos efectos adicionales que pasamos a describir a continuación.

Los pasos que realizamos son los siguientes:

- a. Determinar si la posición del sol se proyecta dentro de la cámara. Para proyectar no solo tenemos que comprobar que el sol está dentro de la visualización de la cámara. También tenemos que lanzar un rayo desde la posición de la cámara al Sol y ver si colisiona con algún objeto.
- b. Si es así, determinar la intensidad del sol en función de su ángulo con respecto a la cámara
- c. Pintar un rectángulo en toda la pantalla con una intensidad proporcional a la del sol y en modo aditivo. De modo que conseguimos un efecto bastante realista cuando miramos directamente al sol.
- d. Pintar el flare en la posición adecuada según lo dicho más arriba.
- e. Pinta un pequeño circulito (*Glow*) en la posición del sol.

El resultado final se muestra en la siguiente imagen.



DESCRIPCIÓN DEL ALGORITMO EMPLEADO PARA RENDERIZAR EL TERRENO

1. Introducción al problema

En el simulador hemos tratado el pintado del terreno de un modo particular con respecto a la generalidad que una malla puede ofrecer. Aunque se podría tratar el terreno como una estructura de malla corriente (es decir una lista de vértices

geométricos y una lista de caras con índices a esos vértices), si ponemos una restricción la estructura de datos a manejar puede ser mucho más sencilla y eficiente para pintar que una malla. La restricción que vamos a imponer es que el terreno es básicamente un mapa de alturas. Es decir, el terreno es una rejilla, tal que para cada punto (x,y) de dicha rejilla le corresponde una altura correspondiente.

En nuestras primeras pruebas empleamos como mapa de alturas una rejilla que se calculaba al azar. Más tarde incorporamos al motor de terrenos la capacidad de poder leer una mapa de alturas de disco. Existen numerosos programas en internet que permiten exportar fácilmente a un mapa de alturas.

Para este proyecto hemos utilizado un mapa de alturas que encontramos en una demo de un videojuego en internet. El juego se llama X-Isle y es un proyecto en desarrollo. Las dimensiones de dicho terreno son 1024 x 1024. Y en cada posición se guarda un flota para indicar la altura. Una primera optimización de espacio posible es reducir ese flota de 32 bits a un entero de 16 bits o menos. Esta optimización se deja para versiones posteriores.

Con unos rápidos cálculos, observamos que si queremos pintar todos los triángulos que representan el terreno, obtenemos la cifra de $(1024 \times 1024 \times 2)$ de 2 millones de triángulos (que en tiempo real suponen unos 60 millones de polígonos por segundo). Esta cifra a día de hoy es imposible de representar con ninguna tarjeta aceleradora. Para solventar estos problemas se emplean técnicas de reducción de detalle (level of detail: **LOD**, a partir de ahora). Con técnicas de LOD, se realizan simplificaciones en la malla según las distancia, de modo que una montaña a varios kilómetros de distancia se pinta con menos triángulos que una que se encuentre a pocos metros.

Nuestro objetivo era desarrollar un algoritmo que permitiera ajustar el nivel de detalle a la velocidad de la máquina (de modo que a más potencia de máquina más potencia visual) y que permitiera visualizar el terreno tanto a nivel de suelo como desde varios kilómetros de altura.

Existen una gran variedad de algoritmos que aceleran el pintado de terrenos, desde algoritmos basados en quadtrees hasta aquellos que emplean técnicas de oclusión. Todos estos algoritmos están orientados a reducir la cantidad de triángulos que se mandan a la tarjeta aceleradora. Ejemplos populares de estos son algoritmos se encuentran en [1] y [2]. Todos estos algoritmos fueron pensados y diseñados mucho antes de que el render por hardware se convirtiera en un standard y por tanto, no son adecuados para obtener el máximo rendimiento. A continuación se describe con detalle el algoritmo implementado para el simulador.

Uno de los requisitos que se debe cumplir para conseguir el máximo rendimiento es intentar paralelizar lo máximo posible el trabajo entre la CPU y la GPU (Graphics Process Unit). De este modo, muchas veces es preferible no aplicar algoritmos de eliminación de polígonos no visibles a la cámara y pintar todos los polígonos por bloques. Es decir, es importante descarga a la CPU del mayor trabajo posible, porque generalmente la GPU esta mucho más optimizada para estas labores.

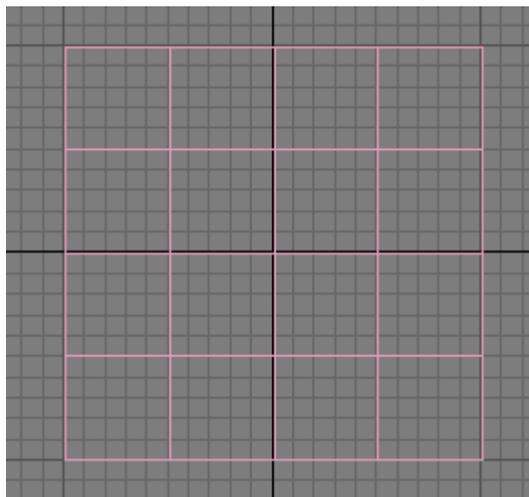
2. Descripción del algoritmo

En esta sección se describe el algoritmo empleado para el simulador.

1. Representación de los datos

El terreno es una malla de puntos con distancias fijas entre ellos. El número de vértices horizontales y verticales en la malla debe ser de la forma $2^n + 1$ ($n > 0$). Esto da un total de 2^{2n} cuadriláteros, con 4 vértices vecinos como esquinas. Cada cuadrilátero consta de 2 triángulos, que son las primitivas que se emplean para pintar.

Cada una de las componentes x e y de un vértice son fijadas a un valor que no varía a lo largo de la simulación. La coordenada z representa la altura del terreno en esa determinada posición. El total del terreno está organizado en bloques que tienen un tamaño prefijado y deben ser de la forma $2^n + 1$ ($n > 0$). Esto es un preproceso para construir el quadtree que será explicado más adelante. Cada uno de los bloques base sirve como nivel 0 de detalle. El nivel 0 representa el máximo detalle. Por ejemplo, en la siguiente figura se muestra un nivel 0 suponiendo que los bloques de división iniciales tengan dimensiones de 5×5 vértices. El tamaño de cada bloque puede elegirse libremente, pero $n = 6$ dio los mejores resultados en nuestras máquinas, y con un tamaño de terreno total de 1024×1024 .



La idea básica del algoritmo es preprocesar los varios niveles de detalle posible. Así, si partimos de bloques de 5×5 , deberíamos calcular los bloques 3×3 y 1×1 .

2. View-Frustum Culling

Debido a que largas secciones del terreno no serán visibles en cada momento (es decir, no estarán dentro de la pirámide de visualización –frustum– de la cámara) para un determinado punto de vista, es necesario eliminar lo antes posible esos bloques para no realizar cálculos con ellos y para que no entren en el proceso de dibujo en pantalla.

En nuestro proyecto nos decantamos por una estructura denominada quadtree, que pasamos a describir.

Los quadtree son una simplificación de los octrees (una descripción detallada puede encontrarse en [3]). Si los octrees trabajan en tres dimensiones, los quadtree son una aplicación al caso particular de las dos dimensiones. Un quadtree es una estructura de datos que basándose en el principio de divide y vencerás, subdivide el primer nivel de terreno en cuatro bloques. A su vez, a cada uno de esos bloques se les va aplicando el mismo proceso, por lo que acabamos teniendo una estructura de árbol que tiene en la raíz todo el terreno (1024 x 1024) y en las hojas el tamaño mínimo de bloque (32x32 en nuestro caso). A su vez, con cada nodo del árbol se guarda una caja que envuelve a ese nodo y a todos sus hijos. El algoritmo ven pseudo-código para aplicar frustum-culling con un quadtree es el siguiente..

```
PintaNodo(n:Nodo)
{
    Obtener BoundingBox de Nodo
    si el BoundingBox esta fuera de la pirámide de visualización de
    la cámara entonces retornar

    Pintar Nodo

    sino
        si no es Hoja

            PintaNodo(n.hijo0);
            PintaNodo(n.hijo1);
            PintaNodo(n.hijo2);
            PintaNodo(n.hijo3);

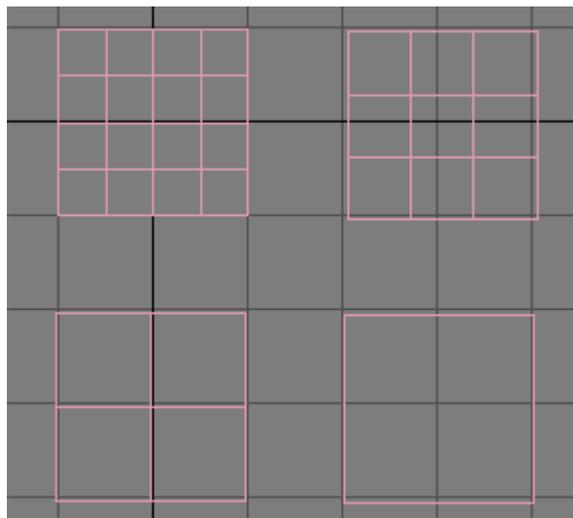
}
```

El quadtree se genera en tiempo de carga. La razón por la que se emplean quadtree en vez de octree es que nuestra estructura de terreno es básicamente una organización espacial en dos dimensiones.

Los bloques de terreno se marcan como visibles cuando sus bounding boxes no se salen de la pirámide de visualización. Una vez que se ha recorrido el árbol, tenemos una lista de bloques de terreno que serán mandados a la tarjeta gráfica una vez elegido su nivel de detalle adecuado. Proceso que se describe en el siguiente punto.

3. Aplicación de la reducción de detalle a los bloques

Los bloques de terreno que están muy lejanos en coordenadas de cámara no necesitan ser renderizados con el mismo detalle que aquellos que están más cercanos a cámara. Se pueden aproximar por una versión con menor resolución (menor número de vértices). Esto es lo que se denomina Level of Detail (LOD). En un precálculo inicial calculamos todos los niveles de detalle de un bloque. Por ejemplo, en la siguiente figura tenemos un ejemplo de dichos niveles.

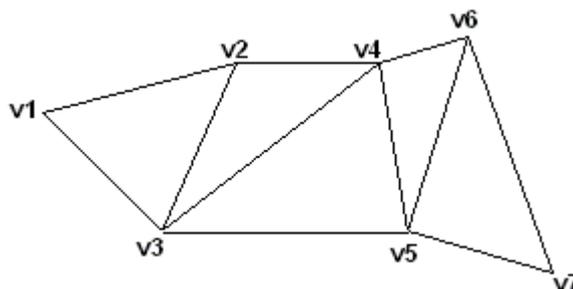


El principal problema a resolver, es como elegir el nivel de detalle adecuado para cada distancia. Cuando pasamos de un nivel de detalle a otro se produce un error en el bloque de terreno debido a la eliminación de vértices en el bloque, que producirá un cambio de altura en algunas aristas. Este cambio será menos visible según la distancia a la cámara aumente, debido a la perspectiva. El cambio producido también se puede proyectar en pantalla y medirse en pixels. De modo que seleccionando un umbral máximo de error podemos calcular para cada bloque de terreno la distancia mínima a partir del cual puede ser mostrado.

De este modo, para cada uno de los bloques visibles resultantes de aplicar el culling con quadtree, se selecciona su nivel de detalle más adecuado en función de su distancia a cámara. Uno de los principales problemas que vamos a encontrar es cómo pintar bloques vecinos que estén a distinto nivel de detalle. Ese punto se discute en la siguiente sección.

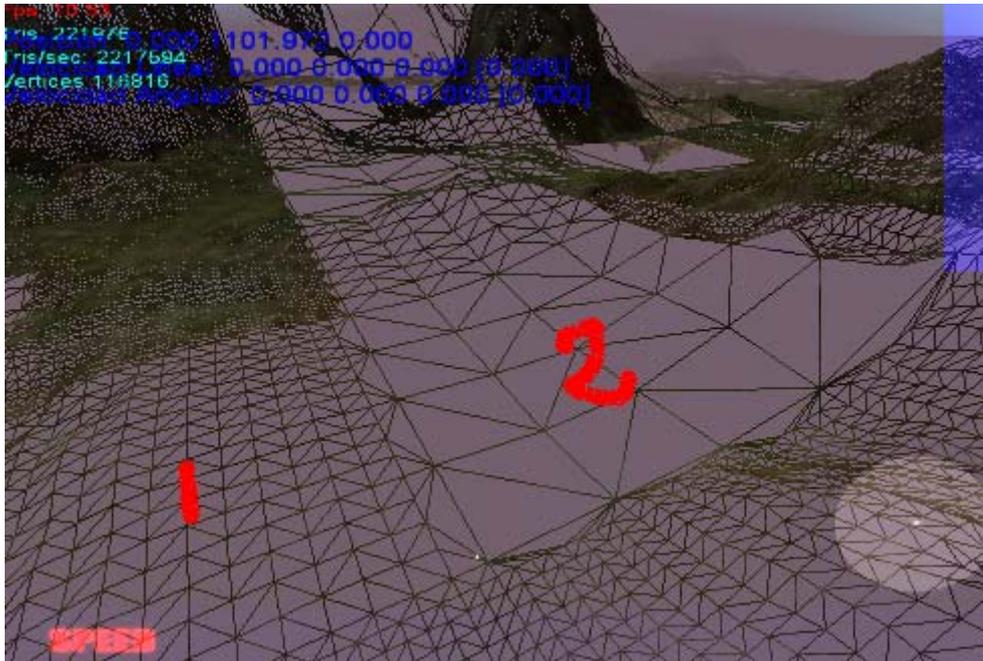
4. Resolviendo el problema de los huecos

A la hora de pintar un bloque de terreno tenemos en cuenta dos casos particulares. Si el bloque no tiene ningún vecino de distinto nivel de detalle que él, entonces se pinta con una tira de strips previamente calculada. Los strips indexados son la primitiva más rápida posible con la que mandar triángulos a la tarjeta gráfica.



En el caso de que el bloque de terreno tenga algún vecino de distinto nivel, entonces el interior del bloque (todo menos el reborde) se pinta con el método anterior de los strips (previamente calculados) y cada uno de los rebordes se “cose” con su vecino. El “cosido” se realiza mediante listas indexadas de triángulos que también son precalculadas para todas las posibles combinaciones.

En la siguiente imagen podemos ver el funcionamiento de este algoritmo. En la foto se han marcado 2 bloques de terreno de distinto nivel (por ejemplo, el bloque marcado con un 2 se pinta con menos polígonos porque tiene una curvatura y una nivelación superior a la que tiene el bloque marcado con un 1). En la figura se observa claramente como se cosen los distintos bloques entre sí.



5. Generando la textura del terreno. Texturas procedurales

Uno de los problemas que se nos planteaba era como texturas el terreno. Para ellos teníamos que tener en cuenta muchas de las posibilidades que nos planteábamos en otras partes del engine.

Necesitábamos un algoritmo que generara texturas casi al azar (proceduralmente), de modo que en función de ciertas condiciones externas pudiéramos conseguir un look u otro. Y el algoritmo debería ser lo más manual posible. Es bastante incómodo tener que texturar a mano el terreno.

Para ello recurimos a un algoritmo que trabaja por capas para generar la textura. Cada capa consta de los siguientes atributos:

- Textura: el nombre del fichero que se emplea para pintar esta capa. La textura es un fichero relativamente pequeño que luego se repite en todas direcciones
- Rango de altura: es el rango de altura (min – max) dentro del cual se activa esta capa.
- Rango de mezcla: son los valores máximos y mínimos de la transparencia de esta capa.
- Angulo de rango: es el rango de los ángulos dentro de los cuales está activa la capa. El ángulo hace referencia a la inclinación del terreno

Para cada píxel del terreno tenemos una serie de capas activas y una serie de factores de mezcla. Aplicando esos factores obtenemos un color final del terreno en cada uno de los píxeles (Nosotros hemos escogido una resolución de 1024x1024 para el total de la textura del terreno).

Todo la gestión de las capas está implementado en la clase *TerrainLayer*. A continuación mostramos el conjunto de texturas empleadas para cada una de las capas.



El siguiente paso era conseguir una iluminación adecuada del terreno. En [17] se describe un algoritmo bastante potente para realizar iluminación en tiempo real. De este modo podemos mostrar una iluminación en función de posición del sol. Aunque nosotros hemos dejado la puerta abierta a dicha implementación, en esta primera versión del simulador la iluminación se precalcula al principio de la ejecución.

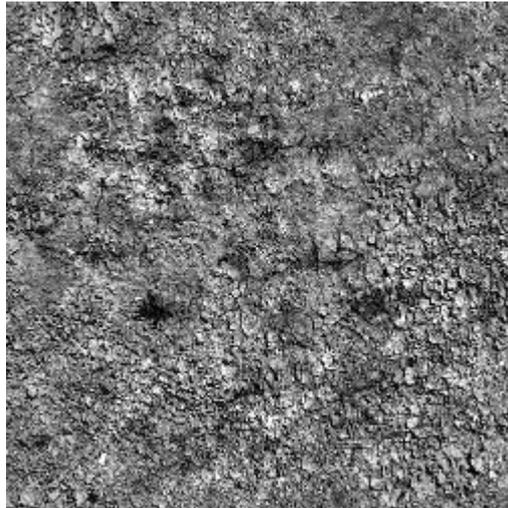
Para calcular la iluminación de un píxel se lanza un rayo desde dicha posición al sol. Si este rayo no interseca con ninguna montaña, entonces dicho píxel se ilumina en función del ángulo que forme con el sol. En otro caso solo recibe el color de la iluminación ambiente.

A continuación mostramos la textura resultante de aplicar el algoritmo descrito en la líneas anteriores.



La textura generada es adecuada cuando estamos viendo el terreno desde una posición relativamente elevada. En el caso de que bajemos mucho al nivel del suelo, la textura empieza a perder resolución. Una posible solución a este problema es emplear una textura de detalle, que se aplica con un factor de repetición muy alto (en nuestro caso la textura de detalle se repite 250 veces dentro de la textura grande de arriba) y que suele pintar filtrando la textura grande.

La textura que hemos empleado nosotros para detalle es la siguiente.



3. Posibles mejoras del algoritmo

Una mejora sustancial vendría de la parte del algoritmo de oclusión. El algoritmo que estamos empleando para eliminar rápidamente bloques no visibles mediante quadtree no es eficiente para casos en que parte del terreno tapen a otras partes. Por ejemplo, si tenemos en frente una montaña, todo lo que quede detrás debería evitarse su pintado. Con nuestra implementación de oclusión esos casos no se tienen en cuenta. La búsqueda de un buen algoritmo de oclusión posiblemente podría llevar el desarrollo de otro proyecto. Una idea básica de aplicación del algoritmo puede encontrarse en [4]. Aunque en [5] encontramos una extensísima documentación sobre técnicas de oclusión.

Otras posibles mejoras que se quedan en el tintero a nivel gráfico, son las siguientes:

- Representación de vegetación. Por ejemplo árboles
- Representación de agua y reflexión
- Representación de sombras. Para tener una idea más clara de la altura a la que esta volando el helicóptero es muy conveniente que este represente sombras sobre el suelo.

4. Rendimiento final obtenido

El algoritmo aquí descrito ha sido fuertemente optimizado para tarjetas aceleradoras bajo la api DirectX v8.0. Se han realizado casos particulares para tarjetas que aceleren la transformación y para tarjetas que no lo aceleren.

La máquina empleada ha sido un PentiumII-350 con una GeForce de 32 Mb. El nivel de detalle seleccionado ha sido aquel que mantenía una media de polígonos en torno a los 250.000. La media de fps ha sido de 20, y el número de polígonos por segundo oscila entre 5 y 6 millones. Sin duda, estamos ante una cifra bastante buena, lo que indica que el algoritmo esta bien optimizado.

5. IMPLEMENTACIÓN DEL MODELO FISICO

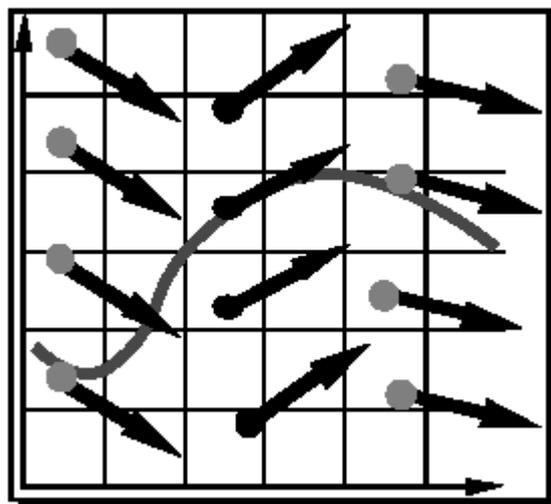
Antes de pasar a describir con detalle el modelo físico implementado en la simulación del helicóptero vamos a explicar algunas nociones básicas para comprender a fondo el funcionamiento del conjunto de ecuaciones empleadas en el desarrollo del proyecto. Primero daremos describiremos brevemente el tipo de ecuaciones diferencias empleadas así como posibles métodos de resolución. En el siguiente apartado repasaremos el conjunto de ecuaciones de sólido rígido que serán necesarias para construir el modelo físico. Finalmente, en el último apartado se describe el modelo implementado.

1. Ecuaciones diferenciales

Una ecuación diferencial describe la relación entre una función desconocida y su derivada. Resolver una ecuación diferencial es encontrar una función que satisfaga la relación (y normalmente alguna condición más). Uno de los problemas en el que nos vamos a centrar es el de los *problemas de valor inicial*. El comportamiento de un sistema se describe mediante una ecuación diferencia ordinaria (ODE a partir de ahora) de la forma:

$$x' = f(x,t)$$

donde f es una función conocida, x es el estado del sistema, y x' es la derivada en el tiempo. Normalmente x y x' son vectores. Como sugiere el nombre, en un problema de valor inicial, tenemos una condición $x(t_0) = x_0$ en un tiempo inicial t_0 . Podemos visualizar el problema en 2D, de modo que $x(t)$ describe el movimiento de un punto p que se dejara en el plano. En cualquier punto x la función f puede ser evaluada para dar un vector un vector de dos dimensiones, de modo que f define un campo vectorial en el plano.



Para resolver una ODE, recurriremos a soluciones numéricas, en las cuales tomaremos pasos en el tiempo empezando con el valor inicial $x(t_0)$. Para avanzar un paso, usaremos la derivada de la función f para calcular un cambio aproximado en x , en un intervalo de tiempo. En el cálculo de la solución numérica, la derivada de la función f es tratada como una caja negra: a partir de unos datos de entrada para x y t , recibimos un valor numérico para x' . Los métodos numéricos trabajan evaluando una o más veces estas derivadas en cada paso de tiempo.

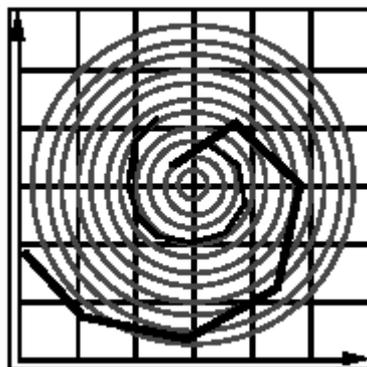
Método de Euler

El método numérico más sencillo es el método de Euler. Denotemos nuestro valor inicial de x por $x(t_0) = x_0$ y nuestra estimación de x en el tiempo $t_0 + h$ por $x(t_0 + h)$, donde h es el delta de tiempo empleado. El método de Euler calcula $x(t_0 + h)$ avanzando un paso en la dirección de la derivada.

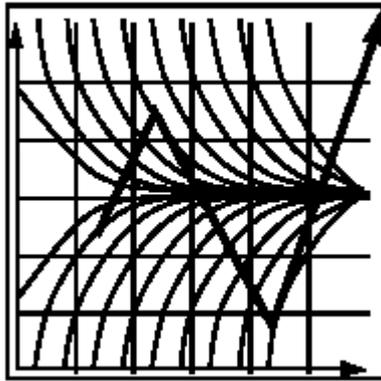
$$x(t_0 + h) = x_0 + hx'(t_0)$$

Teniendo en mente una imagen de un campo vectorial 2D podemos comprender el funcionamiento del método de Euler. En vez de la curva real, p (un punto dejado en el campo vectorial) sigue una ruta poligonal, cuya dirección la determina la evaluación del vector f al principio, y cuyo módulo lo determina h .

Aunque simple, el método de Euler no es preciso. Consideremos el caso de una función f en 2D cuya curva de integración son círculos concéntricos. Un punto p gobernado por f se supone que orbitará para siempre en cualquiera de los círculos en los que empezara. Sin embargo, con el paso de Euler, p se moverá en línea recta hacia un círculo de radio mayor, de modo que la ruta que describirá será una espiral hacia fuera. Haciendo más pequeño el salto de tiempo podemos reducir la apertura de la espiral, pero nunca eliminarla.



Igualmente, el método de Euler puede ser inestable. Consideremos una función 1D $f = -kx$, que debería hacer que el punto p decayera exponencialmente hacia cero. Para pasos de tiempo pequeños obtenemos un comportamiento adecuado, pero cuando $h > 1/k$, tenemos que $|\Delta x| > |x|$, y la solución oscila entorno al cero.



Finalmente, el método de Euler no es eficiente. La mayoría de los métodos numéricos pierden gran parte del tiempo de cálculo calculando derivadas, de modo que el coste por paso se determina según el número de evaluaciones por paso. Aunque el método de Euler, requiere una evaluación por paso, la eficiencia real del método depende del paso de tiempo que empleemos. Métodos más sofisticados, emplean más evaluaciones por paso que el método de Euler, pero permiten emplear unos pasos de tiempo mucho mayores.

A pesar de todas las pegs que hemos puesto al método de Euler, es el que finalmente hemos implementado (aunque la arquitectura del modelo físico permite emplear cualquier método), debido a que para las ecuaciones que nosotros hemos empleado el resultado era bastante bueno. Para una descripción detallada de otros métodos se recomienda [6], y los artículos de [7] y [8] (Estos últimos forman un curso introductorio a métodos numéricos excelente).

2. Simulación de sólido rígido

En esta sección vamos a explicar los conceptos básicos para la implementación de las ecuaciones de movimiento de un sólido rígido.

La simulación de un sólido rígido es muy similar a la simulación de una partícula, de modo que empezaremos con la simulación de una partícula. Denotamos con $x(t)$ la posición de la partícula en coordenadas de mundo en el tiempo t . La función $v(t) = x'(t)$ nos da la velocidad de la partícula en el tiempo t . El estado de una partícula en el tiempo t viene dado por su posición y su velocidad. Generalizamos este concepto definiendo el vector de estado $Y(t)$ para un sistema,

$$Y(t) = \begin{pmatrix} x(t) \\ v(t) \end{pmatrix}$$

Para un sistema de n partículas, generalizamos $Y(t)$,

$$Y(t) = \begin{pmatrix} x_1(t) \\ v_1(t) \\ \vdots \\ x_n(t) \\ v_n(t) \end{pmatrix}$$

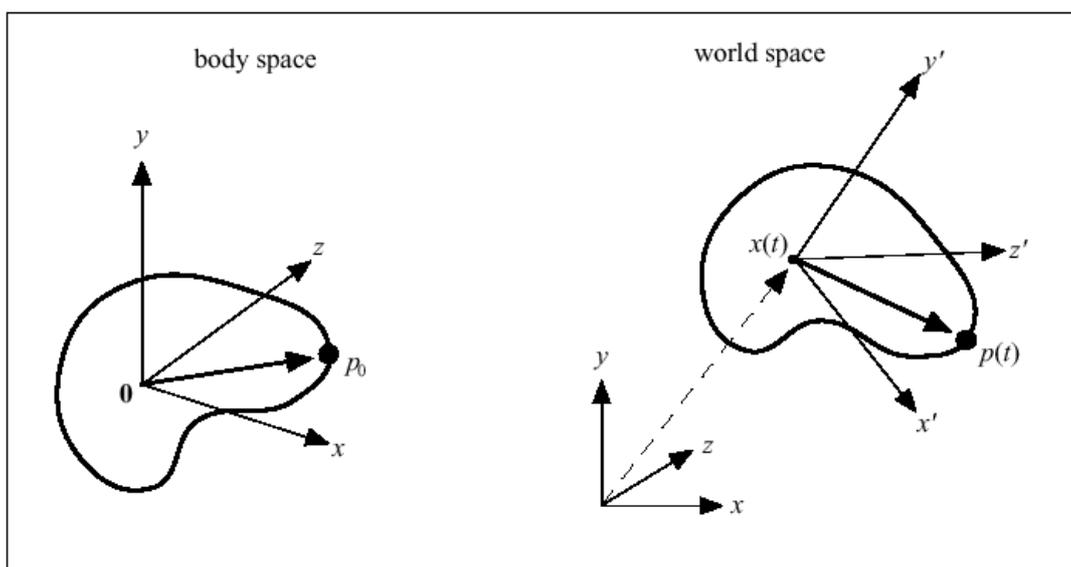
Trabajar con una partícula no es más complejo que trabajar con una, de modo que para simplificar trataremos $Y(t)$ como el vector estado de una partícula.

Dado cualquier valor de $Y(t)$, esta ecuación describe como $Y(t)$ cambia instantáneamente en el tiempo t . Una simulación comienza con una condición inicial $Y(0)$ y luego usa ecuación numéricas para resolver el cambio de Y a lo largo del tiempo. La simulación de un sólido rígido funciona exactamente igual que para las partículas. La única diferencia es que el vector de estado $Y(t)$ de un sólido rígido contiene más información, y su derivada es un poco más compleja.

Conceptos de sólido rígido

La localización de una partícula en el tiempo se puede describir con un vector $x(t)$, que determina la translación de la partícula con respecto al origen. Los sólidos rígidos son algo más complejos, debido a que además de trasladarlos estos pueden ser rotados. Para situar un sólido rígido usaremos un vector $x(t)$ y para su orientación emplearemos una matriz 3x3 de rotación $R(t)$. Llamaremos a $x(t)$ y a $R(t)$ las *variables espaciales* del sólido rígido.

Definimos un sistema local del sólido rígido de tal modo que la posición $(0,0,0)$ de dicho sistema coincide con el centro de masas de dicho sólido rígido. $R(t)$ representa una rotación del sólido alrededor del centro de masas. De modo que un vector fijo r en el sistema local será rotado al vector de mundo $R(t)r$ en el tiempo t .



Debido a que el centro de masas esta fijado en el origen, la posición en mundo de centro de masas viene siempre determinada por $x(t)$. El significado físico de $R(t)$ es que la primera columna de $R(t)$ representa la dirección del eje x del sólido rígido cuando se transforma a mundo en el instante t . Se define similarmente para la segunda y tercera columna.

Velocidad lineal

Definimos la velocidad lineal $v(t)$ como

$$v(t) = x'(t)$$

Si imaginamos que la orientación del sólido esta fija, entonces el único movimiento que puede ocurrir es una translación pura. En este caso $v(t)$ nos da la velocidad de esta translación.

Velocidad angular

Adicionalmente a rotar, un sólido rígido puede girar. Supongamos que impedimos que el centro de masas puede moverse. Entonces cualquier movimiento de los puntos tiene que ser causa de un giro alrededor de algún eje que pase por el centro de masas. Describimos este giro mediante el vector $w(t)$. La dirección de $w(t)$ es la dirección del eje alrededor del cual el sólido esta girando. El módulo de $w(t)$ nos dice lo rápido que está girando. Al vector $w(t)$ se le denomina *velocidad angular*.

Podemos relacionar la velocidad lineal y la velocidad angular con la siguiente expresión:

$$r'(t) = w(t) \times r(t)$$

Del mismo modo que para la translación, la derivada de R vale lo siguiente (esta fórmula se demuestra en [9])

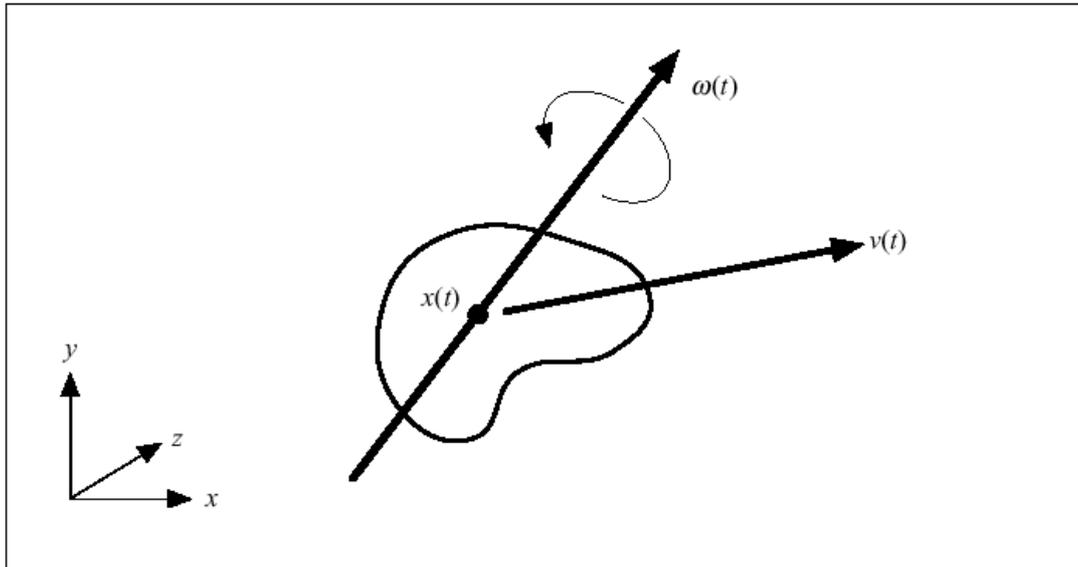
$$R'(t) = w(t)^* R(t)$$

donde a^* representa la matriz

$$\begin{pmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{pmatrix}$$

de modo que

$$a^* b = \begin{pmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{pmatrix} \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} = \begin{pmatrix} a_y b_z - b_y a_z \\ -a_x b_z + b_x a_z \\ a_x b_y - b_x a_y \end{pmatrix} = a \times b$$



Masa total

La masa total de un sólido rígido, es la siguiente suma

$$M = \sum_{i=1}^N m_i$$

Centro de masas

Nuestra definición de centro de masas nos va a permitir separar la dinámica del sólido rígido en componentes lineales y angulares. El centro de masas de un sólido rígido en coordenadas de mundo se define

$$\frac{\sum m_i r_i(t)}{M}$$

donde M es la masa total. Cuando decimos que estamos usando el centro de masas como el origen del sistema de coordenadas local, queremos decir matemáticamente que

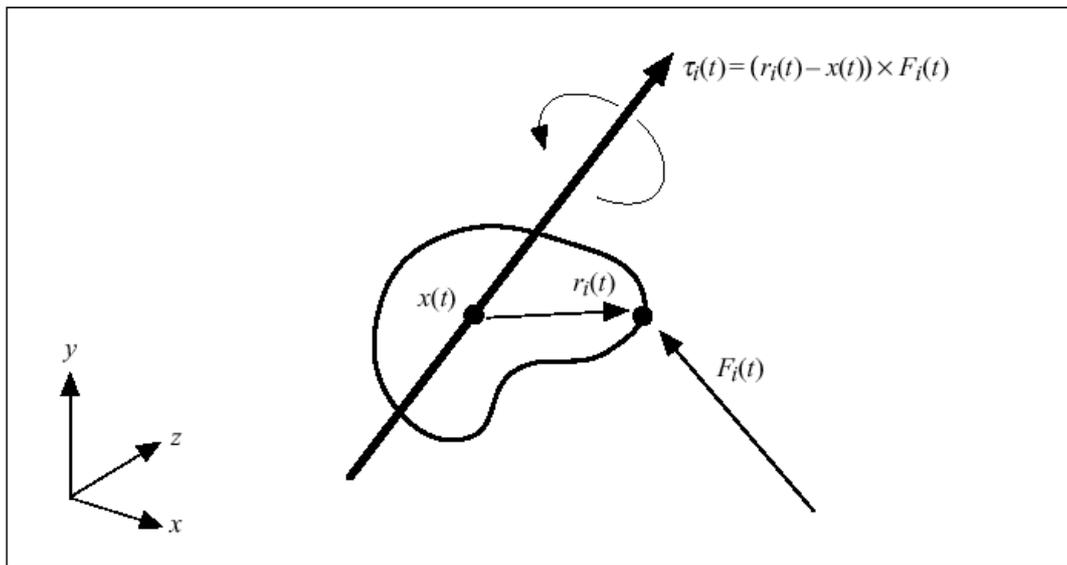
$$\frac{\sum m_i r_{0i}}{M} = 0 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}.$$

Fuerzas y momentos

Cuando pensamos en una fuerza actuando en un sólido rígido debido a una influencia externa (por ejemplo la gravedad), nos imaginamos que la fuerza solo actúa en una partícula del sólido rígido. La localización de esa partícula define el punto en que la fuerza está actuando. Definimos $F_i(t)$ como la suma de las fuerzas externa actuando en la partícula i en el tiempo t. Del mismo modo definimos el momento externo $\tau_i(t)$ actuando en la partícula i como

$$\tau_i(t) = (r_i(t) - x(t)) \times F_i(t)$$

El momento difiere de una fuerza en que el momento de una partícula depende de la posición de la partícula con respecto al centro de masas.



Momento lineal

El momento lineal de una partícula con masa m y velocidad v se define como

$$p = mv$$

El momento lineal total $P(t)$ de un sólido rígido es la suma de todos los momentos de las partículas que forman ese sólido rígido.

Aplicando la primera ecuación de Newton general ($F = m \frac{dv}{dt} = m \frac{d^2x}{dt^2}$)

$$P(t) = \sum m_m v(t) = \left(\sum m_i \right) v(t) = Mv(t)$$

La ecuación anterior nos dice que el momento lineal de un sólido rígido es el mismo que si el sólido rígido fuera una partícula con masa M y velocidad $v(t)$. Gracias a esto podemos obtener la siguiente relación

$$v'(t) = \frac{P'(t)}{M}$$

No lo demostramos aquí, pero en [10] se obtiene la siguiente expresión

$$P'(t) = F(t)$$

Momento angular

Mientras que el concepto de momento lineal es bastante intuitivo, el concepto de momento angular no lo es. Si para el momento lineal tenemos que $P(t) = Mv(t)$, similarmente para el momento angular definimos $L(t)$ como $L(t) = I(t)w(t)$, donde $I(t)$ es una matriz 3x3 llamada tensor de inercia que detallaremos en el siguiente apartado. El tensor de inercia describe como se distribuye la masa en el sólido rígido y la oposición que este muestra a ser rotado. Análogamente a $P'(t) = F(t)$, tenemos

$$L'(t) = \tau(t).$$

El tensor de inercia

El tensor de inercia $I(t)$ es el factor de escalado entre el momento angular y la velocidad angular $w(t)$. El tensor $I(t)$ se expresa mediante la siguiente matriz simétrica

$$I(t) = \sum \begin{pmatrix} m_i(r'_{iy}{}^2 + r'_{iz}{}^2) & -m_i r'_{ix} r'_{iy} & -m_i r'_{ix} r'_{iz} \\ -m_i r'_{iy} r'_{ix} & m_i(r'_{ix}{}^2 + r'_{iz}{}^2) & -m_i r'_{iy} r'_{iz} \\ -m_i r'_{iz} r'_{ix} & -m_i r'_{iz} r'_{iy} & m_i(r'_{ix}{}^2 + r'_{iy}{}^2) \end{pmatrix}$$

Afortunadamente, usando las coordenadas locales del sólido rígido, podemos calcular una única vez un tensor de inercia válido para cualquier orientación $R(t)$. No los vamos a demostrar aquí, pero la siguiente relación es válida (de nuevo recomendamos la lectura de [9])

$$I(t) = R(t)I_{body}R(t)^T$$

I_{body} se calcula una vez al principio a partir de las coordenadas locales del sólido rígido.

Si el sólido rígido es sencillo, entonces su tensor de inercia es fácilmente calculable.

Tensor de inercia de una esfera de radio R

$$I_x = I_y = I_z = \frac{2}{5}mR^2$$

Momento de inercia de un cilindro de radio R y altura h, donde el cilindro esta dispuesto sobre el eje z

$$I_x = I_y = \frac{1}{4}m\left(R^2 + \frac{1}{3}h^2\right)$$

$$I_z = \frac{1}{2}mR^2$$

Momento de inercia de una caja rectangular de lados a, b y c

$$I_x = \frac{1}{12} m(a^2 + b^2)$$

$$I_y = \frac{1}{12} m(a^2 + c^2)$$

$$I_z = \frac{1}{12} m(b^2 + c^2)$$

El momento de inercia de una caja es muy útil, porque podemos aproximar otros cuerpos a la caja que lo contiene.

Ecuaciones de movimiento para el sólido rígido

Tras haber dado una rápida pasada por todos los conceptos necesarios, estamos en condiciones de definir el vector estado $Y(t)$ de nuestro sólido rígido

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

El estado de un sólido rígido es su posición y orientación, y su momento lineal y angular. La masa M de cuerpo y el tensor de inercia local I_{body} son constantes, que asumimos que conocemos al comenzar la simulación. En cualquier instante t , las variables auxiliares $I(t)$, $w(t)$ y $v(t)$ son calculadas de la siguiente manera

$$v(t) = \frac{P(t)}{M}, I(t) = R(t)I_{\text{body}}R(t)^T, w(t) = I(t)^{-1}L(t)$$

La derivada de $Y(t)$ es

$$\frac{d}{dt}Y(t) = \frac{d}{dt} \begin{pmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{pmatrix} = \begin{pmatrix} v(t) \\ w(t) * R(t) \\ F(t) \\ \tau(t) \end{pmatrix}$$

Una pequeña optimización es en vez de representar la orientación del cuerpo del sólido rígido con una matriz $R(t)$ en $Y(t)$, es mejor usar cuaterniones. Brevemente un quaternion es un vector de cuatro elementos que puede ser usado para representar rotaciones. Si sustituimos $R(t)$ en $Y(t)$ por un quaternion $q(t)$, podemos tratar $R(t)$ como una variable auxiliar que es calculada directamente de $q(t)$, del mismo modo que $w(t)$ es calculada a partir de $L(t)$.

El principal problema que tiene usar matrices es la inestabilidad numérica. De modo que según transcurre el tiempo $R(t)$ cada vez representa menos precisamente una

rotación. Esto se puede aliviar mediante el empleo de quaternion, que por otro lado son una representación mucho más compacta.

No vamos a dar aquí una explicación de los quaterniones, pues existe numerosa bibliografía que hacen referencia a ellos. Recomendamos la lectura de [11]. Únicamente mostramos aquí la formula a aplicar para calcular la derivada de un quaternion (demostración en [9]).

$$q'(t) = \frac{1}{2} w(t)q(t)$$

Implementación

Una vez definido el vector de estado $Y(t)$, solo hay que resolver la ecuación diferencia que representa. La clase *OdeSolver* es la clase de la que deben derivar todos los algoritmos de resolución de ODEs. Nosotros para este proyecto hemos obtenido buenos resultados con una implementación del método de Euler.

La clase *OdeSolver* toma como parámetro un vector de estados a través del interface *IdyDy*. Ese interface es implementado por la clase *Helicopter*, donde entra en juego el vector de estado definido en las secciones anteriores.

3. Descripción de un modelo físico de un helicóptero

El principal objetivo que perseguíamos a la hora de diseñar un modelo del comportamiento del helicóptero era conseguir que su funcionamiento fuera totalmente en tiempo real permitiendo una total interactividad con el usuario. Por lo tanto debíamos encontrar un modelo bastante simplificado, pero a la vez lo más aproximado a la realidad para conseguir un realismo adecuado.

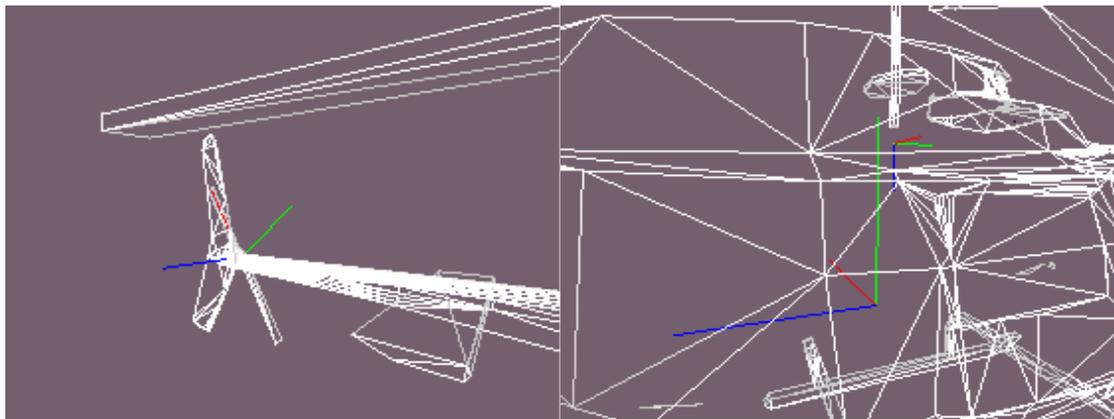
Tras un estudio profundo del modelo matemático descrito en [12] llegamos a la conclusión de que para los requisitos de este proyecto se podrían hacer algunas simplificaciones que acelerarían los cálculos.

Teniendo implementado un integrador de fuerzas y momentos, nuestro modelo lo único que tiene que hacer es proporcionar el conjunto de fuerzas (y su punto de aplicación) que están actuando sobre el helicóptero en cada momento.

Aunque en futuro el modelo debería ser expandible a varios modelos de helicópteros, en esta versión hemos trabajado exclusivamente con el modelo que se muestra en la siguiente imagen.

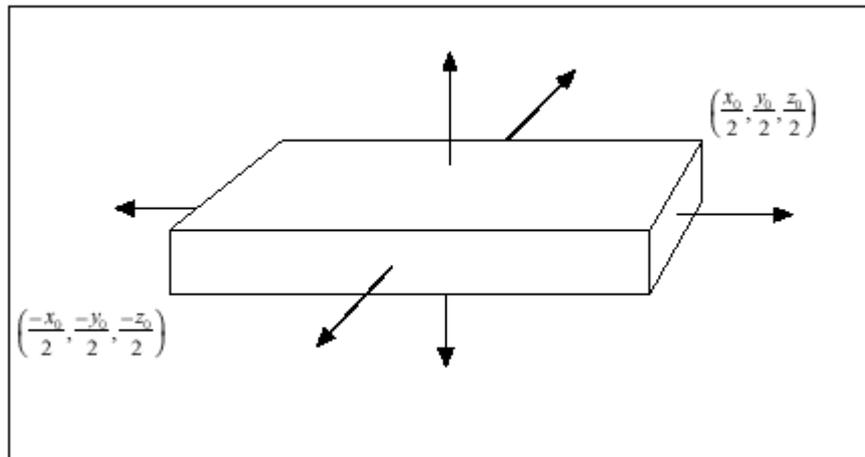


El primer paso a realizar es determinar todos los sistemas locales existentes en el modelo (aquí nos referimos al modelo visual, la malla). En nuestro caso, las únicas tres partes que pueden moverse independientemente las unas con respecto a las otras son los dos rotores y el fuselaje del helicóptero. Mostramos en las siguientes imágenes los ejes locales de cada una de las partes. El color rojo indica el eje X, el verde el eje Y, y el azul el eje Z.



El origen del sistema de referencia del fuselaje (la imagen superior derecha) fue tomando como el centro de masas del total del helicóptero. Aunque dicho centro de masas puede variar a lo largo de la simulación. En nuestras pruebas siempre se considero fijo.

A la hora de calcular el tensor de inercia del helicóptero se realizó la simplificación de considerarlo un bloque rectangular de densidad constante.



De modo que el tensor de inercia (en coordenadas locales del helicóptero) queda del siguiente modo (demostración en [9])

$$I_{body} = \frac{M}{12} \begin{pmatrix} y_0^2 + z_0^2 & 0 & 0 \\ 0 & x_0^2 + z_0^2 & 0 \\ 0 & 0 & x_0^2 + y_0^2 \end{pmatrix}$$

Con todos estos cálculos, pasamos a describir la actuación de cada una de las fuerzas.

Fuerza del rotor principal

El rotor principal produce una fuerza vertical en sentido del eje Z negativo (ver imagen) proporcional a la velocidad de giro. Esa fuerza se traduce en un desplazamiento vertical del centro de masas y de un momento que produce una ligera inclinación del fuselaje en la dirección de avance.

Mediante el cíclico (en nuestro simulador el cíclico se maneja con un joystick, o en su defecto con el teclado) se puede girar el plano de rotación del rotor principal y conseguir desplazar el helicóptero hacia cualquier dirección (lo que suele producir una disminución de la componente vertical que se traduce en una disminución de la velocidad de ascenso o de un descenso si estábamos en vuelo horizontal).

Fuerza del rotor trasero

El rotor trasero está sincronizado con el rotor principal de modo que este cancela el momento que se produce para contrarrestar el giro del rotor principal (por la ley de la conservación del momento angular). Mediante los pedales (que en nuestro simulador se pueden manejar mediante teclado) se puede acelerar o desacelerar el rotor trasero, produciendo un giro en sentido horario o antihorario. Del mismo modo, el giro del rotor trasero produce un ligero desplazamiento en el centro de masas que debe ser compensando adecuadamente.

Fuerza del peso del helicóptero

El peso del helicóptero representa una fuerza que se aplica directamente sobre el centro de masas de este, y es proporcional a su masa. Debido a que el campo gravitacional es uniforme, esta fuerza no produce ningún momento en el centro de masas del helicóptero.

Fuerza de los estabilizadores

En la cola del helicóptero encontramos un estabilizador horizontal y otro vertical. La misión de estos estabilizadores es producir fuerzas de reacción a los giros horizontales y verticales del helicóptero. Hemos modelado la acción de estos elementos añadiendo una amortiguación a la velocidad lineal y angular final (resultante de todo el modelo). En esta amortiguación podemos considerar también incorporada la fricción del aire, que produce una fuerza opuesta proporcional a la velocidad de avance.

Contacto con el suelo

Las colisiones contra el suelo (y las colisiones en general) no están bien resueltas en la versión actual del simulador. Actualmente, cuando el helicóptero colisiona con el suelo (lo que se determina mediante una caja que envuelve al helicóptero) se produce una fuerza opuesta modulo infinito que cancela la fuerza y aceleración actual vertical hacia abajo. Aunque esta solución es bastante pobre, es suficiente para la versión actual del simulador. Versiones posteriores realizarán un tratamiento de colisiones más general.

Básicamente esto se consigue creando una discontinuidad en la función integradora del ODE. La discontinuidad se debe a que la colisión produce un cambio instantáneo de velocidad (debido a un impulso, que es una fuerza que tiene a infinito y que se aplica en un instante de tiempo que tiende a cero). Cuando se detecta una colisión se calcula la nueva velocidad y se continúa con la integración de nuevo.

En [18] tenemos una amplia descripción de las posibles implementaciones de un sistema genérico de colisiones.

Control automático

Con la implementación de modelo descrito en las secciones anteriores el resultado obtenido es bastante bueno, aproximándose bastante a otros modelos matemáticos más complejos.

El principal problema que encontramos es que el control del helicóptero es bastante complicado. Esto es debido a que en un helicóptero, el piloto no tiene que estar atento del conjunto de las fuerzas descritas anteriormente. Existen numerosos mecanismos que automatizan la tarea de conseguir un vuelo equilibrado o un ascenso vertical adecuado. Métodos para implementar dichos mecanismos harían necesaria la inclusión de técnicas de *control digital*.

Aplicar técnicas de *control digital* se sale del ámbito del objetivo del proyecto. Pero nosotros hemos incorporado una pequeña ayuda a la hora de manejar nuestro helicóptero. El principal problema que encontrábamos es que era muy difícil cancelar el momento que hacía que el helicóptero se inclinara hacia delante o hacia atrás.

Debido a esto era complicado mantener un vuelo horizontal y normalmente el helicóptero acaba cayendo en picado hacia el suelo.

El ajuste que hemos realizado afecta al rotor principal. De modo que este no produce una fuerza vertical totalmente hacia arriba, sino que lo produce en una dirección (precalculada, y que podemos considerar dependiente del modelo) tal que cancela el momento de giro del centro de masas. Si nosotros con la palanca de cíclico empujamos, se produce un momento que inclina el helicóptero hacia delante y que hace que avance. En cuanto soltamos la palanca se vuelve a la posición de equilibrio. Lo que se traduce en que hemos conseguido hacer que el helicóptero avance hacia delante sin producirse mas inclinación hacia delante). Este comportamiento es exactamente el que se produce en un helicóptero de verdad. Con este ligero ajuste al modelo el comportamiento del helicóptero se vuelve mucho más sencillo.

6. DESCRIPCIÓN DE LAS POSIBLES MEJORAS

Sistema Multiplayer

Una de las ampliaciones más interesantes que se pueden realizar es la de crear un sistema de juego on-line, en el que diferentes usuarios pudieran interactuar en el mismo entorno.

Este sistema se está imponiendo en los últimos años dada la rápida expansión de Internet, y ya ha dado sus frutos, de tal manera que la mayoría de los sistemas de simulación incluyen como uno de sus principales componentes el de juego on-line.

Dado que este simulador evolucionaría de una manera natural hacia ese campo, nos ha parecido atractivo realizar un intenso estudio sobre su posible implementación. De hecho, las bases están sentadas para que en un futuro el simulador se aproveche de este módulo.

A continuación pasamos a mostrar una primera visión general de este tipo de sistemas, para comentar el que nosotros hemos elegido, y finalmente incluir algunas notas sobre lo que es la implementación en sí del sistema.

Arquitecturas de diseño de un sistema de juego en red

Síntesis

Objetivo

Exponer las diferentes opciones disponibles a la hora de diseñar un sistema multiusuario por red.

Descripción

Este documento describe las principales arquitecturas para una red multiusuario, la arquitectura Cliente-Cliente y Cliente-Servidor.

Desarrollo

Arquitectura

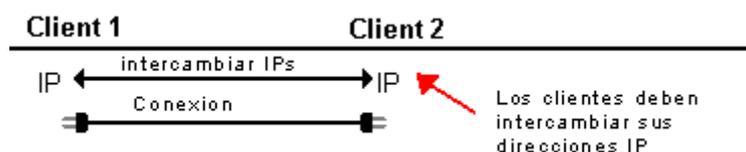
Uno de los puntos a tener en cuenta al realizar la implementación de un sistema multiusuario en red es la arquitectura a utilizar. Las dos principales arquitecturas son cliente-cliente y cliente-servidor.

Cliente - Cliente

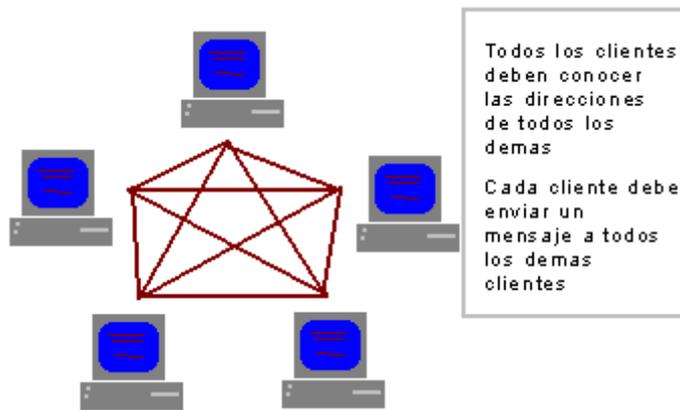
La arquitectura cliente-cliente (también llamada peer-to-peer) es bastante simple, aunque debería ser usada solo para conexiones de dos usuarios solo. Dos clientes se conectan el uno al otro usando la dirección IP del otro o un dominio/host name. El problema que encontramos en esta conexión es que la IP de cada cliente suele cambiar en cada conexión si se trata de clientes a través de modem.

Otra opción es utilizar un servidor que este constantemente conectado para que los clientes puedan encontrar IP's de otros usuarios con facilidad. Esto significa que las sesiones son privadas y de dos usuarios. Las dos máquinas intercambiarán los estados de la simulación y cada una realizará todo el proceso, malgastando recursos.

El diagrama de esta arquitectura es el siguiente:



Y para varias máquinas:



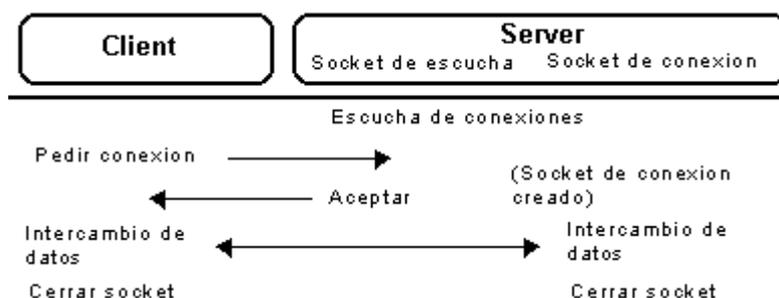
Cliente - Servidor

Esta es la arquitectura correcta. Una maquina con una conexion rapida a Internet o a una red local inicia una aplicación de servidor. Todo funciona a traves de este servidor y hace todo el proceso para todos los clientes (que reduce el proceso si se trata solo de una seccion). El servidor puede o no procesar los datos, dependiendo de cómo se programe. Lo importante es que (teoricamente) el servidor puede soportar un numero ilimitado de clientes que mostraran lo que el servidor les envio.

Otro tema importante es que el servidor no se suele lanzar en una máquina privada, sino en servidores especiales dedicados, con un nombre de dominio para que no se tenga que conocer la direccion IP, o cambiarla.

Además, normalmente estos servidores ofrecen lobbies donde se puede encontrar a otros jugadores, chatear, esperar...

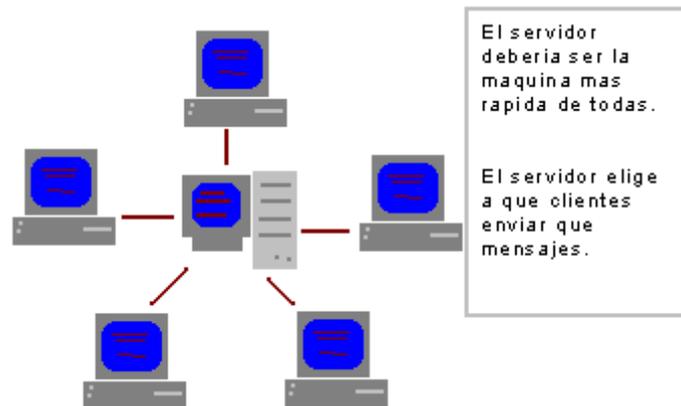
Este es un diagrama de la arquitectura cliente-servidor:



Hay tres sockets en este diagrama. Esta el socket cliente (maquina cliente), y dos usados por el servidor. Al principio solo esta el socket de escucha, que espera en un puerto definido (que el cliente conoce). Cuando este acepta una conexión, crea un nuevo socket, el socket de conexión. Esto es logico, porque si el cliente usara todo el

tiempo el socket de escucha para comunicarse e intercambiar datos con el servidor, entonces nadie mas podria conectarse durante ese tiempo.

Este es un diagrama de varias maquinas conectadas bajo la arquitectura cliente-servidor:



El problema con Internet y a veces incluso con redes locales es que frecuentemente son demasiado lentas. Eso significa que es importante que haya un balance entre el tamaño de paquete y la velocidad. Dependiendo del caso, el servidor puede hacer mas o menos.

En algunos casos se usan tambien arquitecturas hibridas, donde hay un servidor, pero los clientes no tienen que conectarse a traves del servidor. El camino para mantener las conexiones simples, rapidas y eficientes implica el uso de una arquitectura cliente-servidor.

Modelo teórico de un sistema cliente-servidor

Síntesis

Objetivo

Mostrar cual sería la arquitectura real de un sistema cliente-servidor, y compararlo con otros métodos que ya se han implementado

Descripción

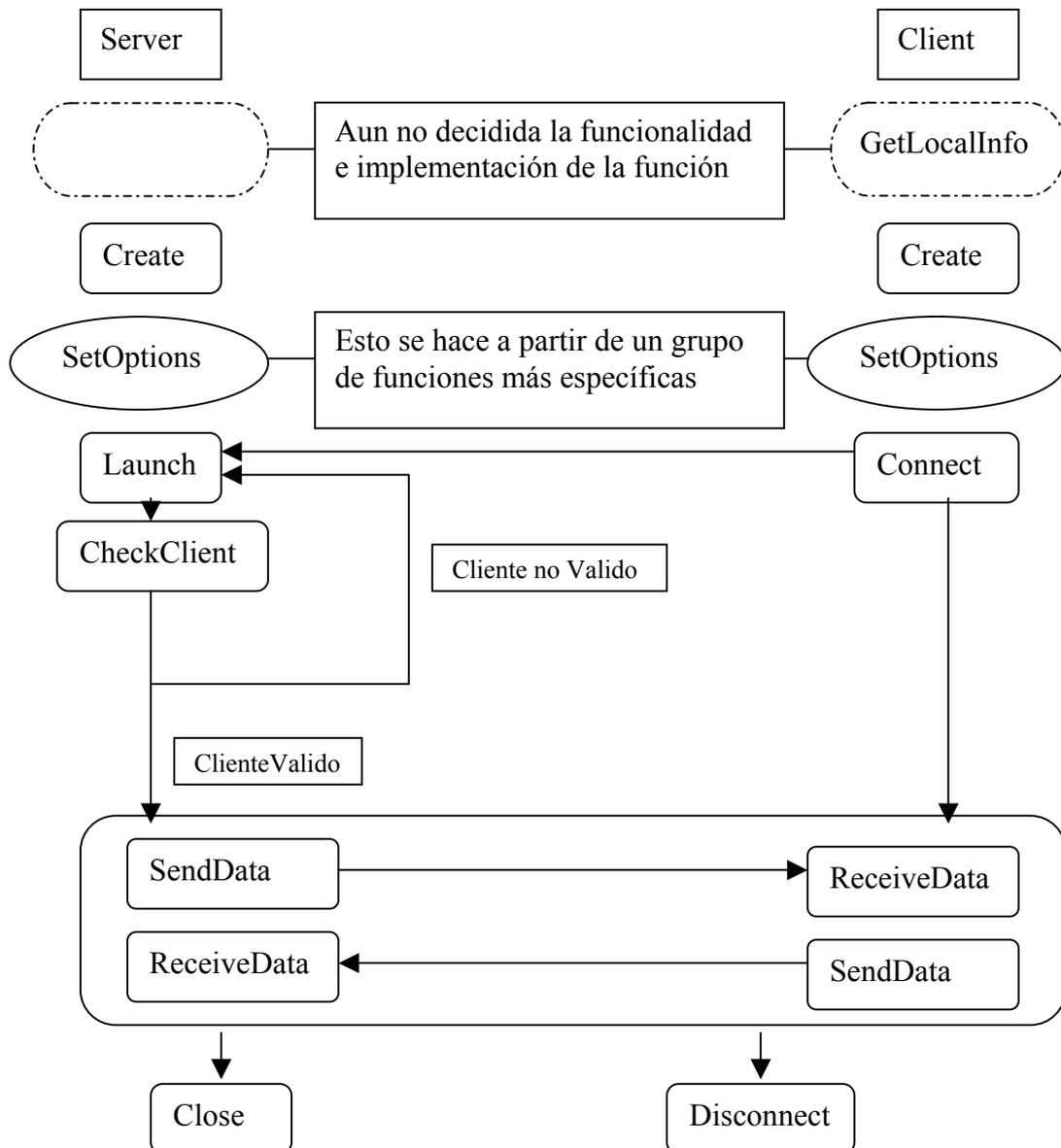
Este documento describe una posible implementación de un modelo cliente-servidor para un entorno multiusuario en una simulación en la que se deban transmitir pequeñas cantidades de datos.

Comparación con otros métodos utilizados de manera estandar: Sockets en Java y en las MFC de Microsoft.

Desarrollo

A continuación vamos a exponer las funciones más comunes que se necesitarían.

La implementación del interfaz de comunicación podría ser la siguiente:



La interfaz de la clase que se utiliza para enviar datos constaría de un constructor al que se le pasa como parámetro los datos a encapsular, y métodos para obtener y especificar estos.

Esto se puede implementar con un sistema de **Timeout** para evitar que el cuelgue, o la falta de transmisión de datos por parte de un cliente siga ocupando espacio en el servidor. Esta opción se configuraría con un método que especificara el

tiempo de Timeout, siendo el proceso transparente al programador, y que formaría parte del conjunto de funciones usadas para especificar las opciones del servidor.

Asimismo, mediante otro método podemos configurar si el servidor será de **tipo** UDP (envío y recepción de datagramas, sin establecer conexión) o TCP (establecimiento de una conexión).

En esta última opción se debe implementar en el Servidor la capacidad de aceptar **diferentes conexiones** de diferentes clientes.

El envío de datos se puede hacer a través de una clase **StreamBuffer**, con dos miembros, un buffer de datos, y el tamaño de este buffer. Se debe exigir que lo que se quiera enviar sea serializable, sino las referencias (punteros) no se podrán enviar. Para esto se puede implementar un método en esta clase que serialice todo lo que se meta en el buffer de forma genérica, aunque parece más práctico exigir que los datos que se inserten en el buffer estén ya serializados.

Comparación con otros métodos

Los métodos más significativos que pasamos a comentar son el uso en Java y en las MFCs de los sockets.

Java:

En Java se usan dos clases, `ServerSocket` (Servidor) y `Socket` (Cliente). De este modo cada clase tiene funcionalidades propias de su finalidad, tal y como nosotros hacemos. El método de conexión cliente-servidor es muy parecido al aquí implementado, con ciertas particularidades. Así, para enviar los datos, se usa directamente una sobrecarga de los operadores “<<” y “>>”, de tal manera que con ellos se direccionan los datos a enviar directamente al socket abierto.

El método utilizado para permitir la conexión simultánea de distintos clientes es dejarle al programador esta responsabilidad, que a grandes rasgos se podría implementar de la siguiente manera:

```
while (true) {  
    accept a connection ;  
    create a thread to deal  
    with the client ;
```

Se permite el *broadcasting*, usando datagramas como método de envío. En modo UDP, se proporciona la creación de un *socket Multicast*, que se usa principalmente para que el servidor envíe datos a un grupo de clientes que se puede especificar. Sería interesante implementar esto en nuestro protocolo.

En esta librería, como hemos dicho antes, es el programador el que tiene que hacer el servidor multithread, ya que no está implementado directamente en la

librería. En nuestro caso no está planteado de esta manera, ya que supone que la librería está creada a muy bajo nivel, como con la clase *CAsyncSocket* de las *MFCs*, que veremos más adelante. Así, realizarlo de esta manera supondría un cambio importante en la arquitectura creada hasta ahora en el interfaz.

Esto proporciona más flexibilidad al programador, con lo que se puede pensar en adoptar una solución como en las *MFCs*, que no es más que crear una clase a bajo nivel (como esta) y otra a más alto nivel (que es como se está desarrollando en nuestro caso).

Tal y como está planteado el caso en este momento, la librería permite la creación de un servidor y un cliente, con envío y recepción de datos, de tal manera que el programador se olvide de los sockets y tan sólo envíe y reciba datos a su antojo, desde donde sea, y de los puestos que sea, lo cual lo hace menos flexible, pero mucho más sencillo a la hora de usarlo.

MFCs:

Respecto a las *MFCs*, su organización se basa en dos clases:

- *CAsyncSocket*, que no es más que una abstracción de las funciones de la librería *Winsock* de windows, pero con las ventajas de estar orientada a objetos, y con la posibilidad del control de errores que facilita, mucho más completo que con las funciones *Winsock*
- *CSocket*, que hereda de *CAsyncSocket*. Es una clase del tipo del sistema que nosotros estamos siguiendo, es decir, facilitando al programador menos control sobre los detalles del protocolo a bajo nivel. Para la transferencia de datos utiliza un objeto del tipo *CArchive*, que se asocia al socket para poder utilizarlo como puente para el envío y la recepción de datos.

La clase *CArchive*, que como hemos dicho es la que se utiliza para la transferencia de datos, tiene la capacidad de ser serializada. Esto es un requisito indispensable para todo lo que se envíe a través del protocolo.

Como en Java, en las *MFCs* se utiliza la recarga de los operandos “<<” y “>>” para la colocación de datos en el buffer de entrada y de salida, dependiendo del caso.

Entre los miembros de la clase *CAsyncSocket*, destacan los métodos de recoger información del socket que usamos:

- “*GetSockOpt*” permite obtener las opciones del socket, tales como el tamaño del buffer de envío y recepción, el status, el envío de señales de keep-alive...
- “*GetSockName*” permite obtener los datos del socket local
- “*GetLastError*” devuelve el último error que se ha producido en el Socket

El resto de métodos son una encapsulación de las funciones de Winsock (accept, connect, bind, listen...), con lo que la programación es muy parecida en ambos casos.

Sockets para windows (Winsock)

Síntesis

Objetivo

Mostrar paso a paso la implementación de un sistema multiusuario por red bajo Windows en base a bibliografía básica.

Descripción

Este documento muestra los diferentes pasos a seguir para realizar una implementación basada en los sockets para Windows de un sistema de comunicación entre programas que permita el envío de la información necesaria para llevar a cabo una simulación en ordenadores conectados en red.

Desarrollo

Winsock

Hay dos versiones de WinSock, version 1.1 y version 2. Se recomienda siempre usar la version 2, para no usar directamente TCP/IP.

Para empezar, se debe añadir un *#include* al comienzo de cualquier archivo que utilice Winsock. Asimismo, se debe añadir la librería *ws2_32.lib* a cualquier proyecto que utilice Winsock.

Hay diferentes maneras de programar con WinSock. Se pueden usar las funciones básicas de UNIX/Berkley, o la version especializada para Microsoft Windows, o incluso la version orientada a objetos de las MFC's.

Las funciones especializadas de Windows son muy útiles, pero obligan a realizar una aplicación de Win32, permitiendote “engancher” los sockets a mensajes propios de Windows que se enviaran al programa. En caso de hacer un servidor, esto no es cierto. Para hacer un servidor se recomienda el uso de las funciones básicas de UNIX/Berkley.

Tipos:

sockaddr

Descripción: *sockaddr* se usa para especificar una conexión de sockets. Se debe usar siempre que sea posible, ya que esta orientado a TCP. Este tipo de datos se usa para guardar informacion sobre un socket (numero de puerto, direccion IP, etc.) que acepta el servidor. Solo un servidor usa este tipo de datos.

sockaddr_in

Descripción: sockaddr_in se usa para especificar una conexión de sockets. Contiene un campo para especificar una dirección IP y un puerto. Esta versión de sockaddr está orientada a TCP. Se usará este tipo cuando se creen sockets.

```
struct sockaddr_in
{
    short sin_family;    // Protocol type (should be set to AF_INET)
    u_short sin_port;    // Port number of socket
    struct in_addr sin_addr; // IP address
    char sin_zero[8];    // Unused
};
```

WSAData

Descripción: WSAData se usa cuando se carga e inicializa la librería ws2_32.dll. Esta estructura de datos se llena llamando a la función WSASStartup (). Se usa para determinar si el ordenador está ejecutando la versión correcta de WinSock.

SOCKET

Descripción: SOCKET es el tipo de datos usado para guardar manejadores de sockets. Estos manejadores se usan para identificar un socket. El SOCKET realmente no es más que un unsigned int.

Conceptos basicos

Para cargar ws2_32.dll emplearemos este código:

```
// Realizar al comienzo de cada programa
WSADATA w; // Usado para guardar la version de Winsock
int error = WSStartup (0x0202, &w); // Relleno de w

if (error)
{ // Hubo un error
  return;
}
if (w.wVersion != 0x0202)
{ // versión incorrecta de Winsock
  WSACleanup (); // Descargar ws2_32.dll
  return;
}
```

0x0202 significa version 2.2. Si se necesitara la version 1.1, se debria cambiar por 0x0101. WSStartup () rellena la estructura WSADATA y carga la librería dinámica de WinSock2. WSACleanup() descarga la WinSock DLL.

Creación de un socket:

```
SOCKET s = socket (AF_INET, SOCK_STREAM, 0); // Crear socket
```

Esto es todo lo que se necesita para crear un socket, pero hablar que hacer un “bind” al puerto despues, cuando se quiera realmente usar. AF_INET es una constante definida en winsock2.h. Si alguna función necesita informacion sobre la familia de direcciones a utilizar, esta será AF_INET. SOCK_STREAM es una constante que le dice a Winsock que necesitamos un socket de tipo stream (TCP/IP). Tambien se puede tener un socket de tipo datagrama (UDP), pero son menos seguros. Dejando el ultimo parametro a 0 seleccionará el protocolo correcto, que deberia ser TCP/IP).

Para realmente asignar un puerto a un socket:

```
// Hacemos bind sobre sockets de servidor, no de cliente
// SOCKET s es un socket valido
// WSASStartup se ha llamado

sockaddr_in addr; // la estructura de la direccion TCP/IP

addr.sin_family = AF_INET; // Familia de direcciones de Internet
addr.sin_port = htons (5001); // Asignar el puerto 5001 al socket
addr.sin_addr.s_addr = htonl (INADDR_ANY); // Sin destino prefijado
if (bind(s, (LPSOCKADDR)&addr, sizeof(addr)) == SOCKET_ERROR)
{ // error
  WSACleanup (); // Descargar WinSock
  return; // Salir
}
```

Esto puede parecer confuso, pero no lo es tanto. *addr* describe el socket especificando el puerto. Establecemos la dirección a INADDR_ANY, que permite que esta sea cualquier dirección, ya que en el fondo no nos importa de donde venga la conexión.

Usamos htons () and htonl () para convertir short y long, respectivamente al formato correcto para que la red lo comprenda. Si tenemos un numero de puerto 7134 (short), entonces usamos htons (7134). Usaremos htonl () en la dirección IP. Pero si queremos especificar la dirección IP no usaremos htonl (), sino inet_addr (). Por ejemplo inet_addr ("129.42.12.241"). inet_addr obtiene la cadena de caracteres y quita los puntos (".") y lo convierte en un long.

Para escuchar en el puerto dado:

```
// WSASStartup () ha sido llamado
// SOCKET s es valido
// s ha sido conectado al puerto usando sockaddr_in sock
if (listen(s,5)==SOCKET_ERROR)
{ // error! incapaz de escuchar
  WSACleanup ();
  return;
}

// escuchando...
```

Ahora lo que falta es aceptar la conexión de algún cliente que se intente conectar. La única peculiaridad sobre el código anterior es en listen (SOCKET s, int backlog). Backlog indica el número de clientes que se pueden conectar al socket mientras esta en uso. Esto significa que esos clientes tendrán que esperar hasta que todos los clientes anteriores hayan sido tratados. Si se especifica un backlog de 5 y

siete clientes tratan de conectar, entonces los últimos dos recibirán un mensaje de error diciendo que deben intentar conectarse más tarde. Normalmente un backlog de entre 2 y 10 es bueno, dependiendo de cuántos usuarios espera el servidor.

Para conectarse a un socket:

```
// WSStartup () ha sido llamado
// SOCKET s es valido
// s ha sido conectado al puerto usando sockaddr_in sock
sockaddr_in target;

target.sin_family = AF_INET;
target.sin_port = htons (5001); // indica el numero de puerto del servidor
target.sin_addr.s_addr = inet_addr ("52.123.72.251"); // indica la IP del
servidor

if (connect(s, target, sizeof(target)) == SOCKET_ERROR)
{ // error
  WSACleanup ();
  return;
}
```

Esto es todo lo que hay que hacer para pedir una conexión. target obviamente define el socket al que intentamos conectar. La función connect () requiere un socket válido (s), la descripción del socket destino (target), y el tamaño o longitud de la descripción (sizeof(target)). Esta función enviará una petición de conexión y después esperará a ser aceptada o informará de cualquier error.

Aceptar una conexión:

```
// WSAStartup () ha sido llamado
// SOCKET s es valido
// s ha sido conectado al puerto usando sockaddr_in sock
// s esta escuchando

#define MAX_CLIENTS 5;

int number_of_clients = 0;
SOCKET client[MAX_CLIENTS]; // socket manejador de clientes
sockaddr client_sock[MAX_CLIENTS]; // info de los sockets clientes

while (number_of_clients < MAX_CLIENTS)//deja conectar a
MAX_CLIENTS
{
    client[number_of_clients] = // acepta una conexion
        accept (s, client_sock[number_of_clients], &addr_size);
    if (client[number_of_clients] == INVALID_SOCKET)
    { // error
        WSACleanup ();
        return;
    }
    else
    { // client conectado
        // iniciamos un thread que se conectara con el cliente
        startThread (client[number_of_clients]);
        number_of_clients++;
    }
}
```

MAX_CLIENTS no es realmente necesario, pero lo usamos para hacer el código más limpio. number_of_clients es un contador que mantiene cuantos clientes están conectados. client[MAX_CLIENTS] es un array de SOCKETS que es usado para guardar los manejadores de los sockets que estan conectados a los clientes. client_sock[MAX_CLIENTS] es un array de sockaddr que se usa para mantener información sobre el tipo de conexión, puerto, etc.

Normalmente no intervenimos en client_sock, pero bastantes funciones lo requieren como parametro. Basicamente este bucle espera hasta que alguien pide una conexión, la acepta y lanza un thread que se comunicará con el cliente.

Escribir (o enviar):

```
// SOCKET s esta inicializado
char buffer[11]; // buffer de 11 caracteres
sprintf (buffer, "Whatever...");

send (s, buffer, sizeof(buffer), 0);
```

El segundo parametro de send () es const char FAR *buf y apunta al buffer de chars que queremos enviar. El tercer parametro es un int con la longitud del buffer que queremos enviar. El ultimo parametro es para flags que no usaremos, lo dejamos en 0.

Leer (o recibir)

```
// SOCKET s esta inicializado
char buffer[80]; // buffer de 80 caracteres

recv (s, buffer, sizeof(buffer), 0);
```

recv () es bastante parecido a send, solo que esta vez no transmitimos un buffer, sino que lo recibimos.

Resolvemos una direccion IP o una URL

```
// const char *Host contiene bien una direccion IP o un nombre de dominio
u_long addr = inet_addr(Host); // Intenta identificar una direccion IP
if (addr == INADDR_NONE) {
    // El host no es una IP, usamos un DNS
    hostent* HE = gethostbyname(Host);
    if (HE == 0) {
        // error
        WSACleanup ();
        return;
    }
    addr = *((u_long*)HE->h_addr_list[0]);
}
```

Lo que hace el codigo es convertir const char *Host en u_long addr, que podemos usar para conectar a otro equipo.

Para cerrar un socket:

```
shutdown (s, SD_SEND); // s no puede enviar en adelante  
  
// aqui deberiamos chequear que no llegan mas datos  
  
closesocket (s); // close
```

shutdown(SOCKET s, int how) bloquea un atributo específico del socket. Los posibles atributos (que son pasados en el parámetro how):

- SD_SEND significa que el socket no puede enviar mas
- SD_RECEIVE significa que el socket no puede recibir mas
- SD_BOTH significa que el socket no puede enviar o recibir

Sockets bloqueantes, no-bloqueantes y asíncronos

Por el momento hemos hablado de sockets bloqueantes. Cuando hacemos una llamada a `accept ()`, la función sólo espera por una petición de conexión o un error. Este es el tipo de socket que se crea por defecto con una llamada al comando `socket ()`.

El siguiente tipo de socket es el no-bloqueante. Con un socket no-bloqueante, las funciones como `accept ()` regresan inmediatamente después de haber sido llamadas, devolviendo bien un error, un resultado positivo, o nada (queriendo esto decir que el resultado llegará después). Estos sockets son computacionalmente ineficientes porque deberemos crear una serie de bucles que esperen a que algo ocurra. Así, no es bueno en la práctica usar este tipo de sockets. Los sockets no-bloqueantes se crean con `select ()`.

Los sockets asíncronos son específicos de Win32. Estos sockets, como los no-bloqueantes, regresan inmediatamente. La diferencia está en que le das a la ventana de configuración una función con mensajes de windows que se deben enviar cuando el evento ocurra.

Así, quedaría:

```
#define WM_ONSOCKET WM_USER+1

...manejador de mensajes...

case WM_ONSOCKET:
{
  if (WSAGETSELECTERROR(lparam))
  {
    // error
    WSACleanup ();
    return 0;
  }
  switch (WSAGETSELECTEVENT(lparam))
  {
    case FD_READ: // datos recibidos
      ...
    case FD_CONNECT: // conexion aceptada
      ...
  }
} break;
```

Esto esta bien si usas MFC o la API de Win32. Puedes hacer otras cosas coo dibujar graficos, recibir entrada del usuario... mientras esperas al evento del socket.

Para aplicaciones servidoras los sockets bloqueantes son los más lógicos y simples, además de practicos, porque todo lo que quieres es esperar a una conexión y entonces comunicarte con el cliente.

Para aplicaciones cliente, usaremos sockets asíncronos, por ser los mas eficientes cuando quieres hacer otra cosa que esperar mientras consumimos tiempo de CPU.

Usando sockets asíncronos

Este es el prototipo de la función en la que estamos interesados:

```
int WSAAsyncSelect (SOCKET s, HWND hWnd, unsigned int wMsg, long lEvent);
```

hWnd es el manejador de la ventana de la aplicación (que será el manejador de la ventana principal). wMsg es el mensaje que quieres enviar a la aplicación cuando el evento deseado ocurre. lEvent especifica el evento que causará que WinSock envíe el mensaje wMsg a la aplicación. Estos son algunos de los flags que se pueden usar:

FD_READ	- Recibiendo datos
FD_WRITE	- El socket está listo para enviar datos
FD_CONNECT	- El servidor ha aceptado y estamos listos
FD_CLOSE	- El socket ha cerrado

Se puede hacer OR de los que queramos usar. Estos no son todos los flags que podemos usar, pero si los que son mas útiles para aplicaciones cliente.

Cuando añadimos código para manejar un nuevo socket con un evento en el manejador de mensajes de nuestra aplicación, se necesitarán conocer dos macros. WSAGETSELECTERROR () se usa para averiguar que ocurrió después de un error. WSAGETSECTEVENT () se usa para encontrar que evento “disparó” el mensaje.

```
#define WM_ONSOCKET WM_USER+1

...SOCKET s esta inicializado. Lo colocamos en modo asincrono...

WSAAsyncSelect (s, hWnd, WM_ONSOCKET, (FD_READ |
FD_CONNECT | FD_CLOSE));

...en el manejador de eventos...

case WM_ONSOCKET:
{
if (WSAGETSELECTERROR(lparam))
{ // error
WSACleanup ();
return 0;
}

switch (WSAGETSECTEVENT(lparam))
{
case FD_READ:
...recibir datos ...
case FD_CONNECT:
...empezar a enviar datos ...
case FD_CLOSE:
...salir ...
default:
...no hacer nada ...
}
} break;
```

Solo los parametros lparam y wparam son usados.

Chequeo de errores

El chequeo de errores es molesto, pero necesario. La mayor parte de las funciones de WinSock devuelven un entero. Como regla general, se puede decir:

```
if (function(...) == SOCKET_ERROR)
{
    cout << "Error!\n";
    WSACleanup ();
    return;
}
```

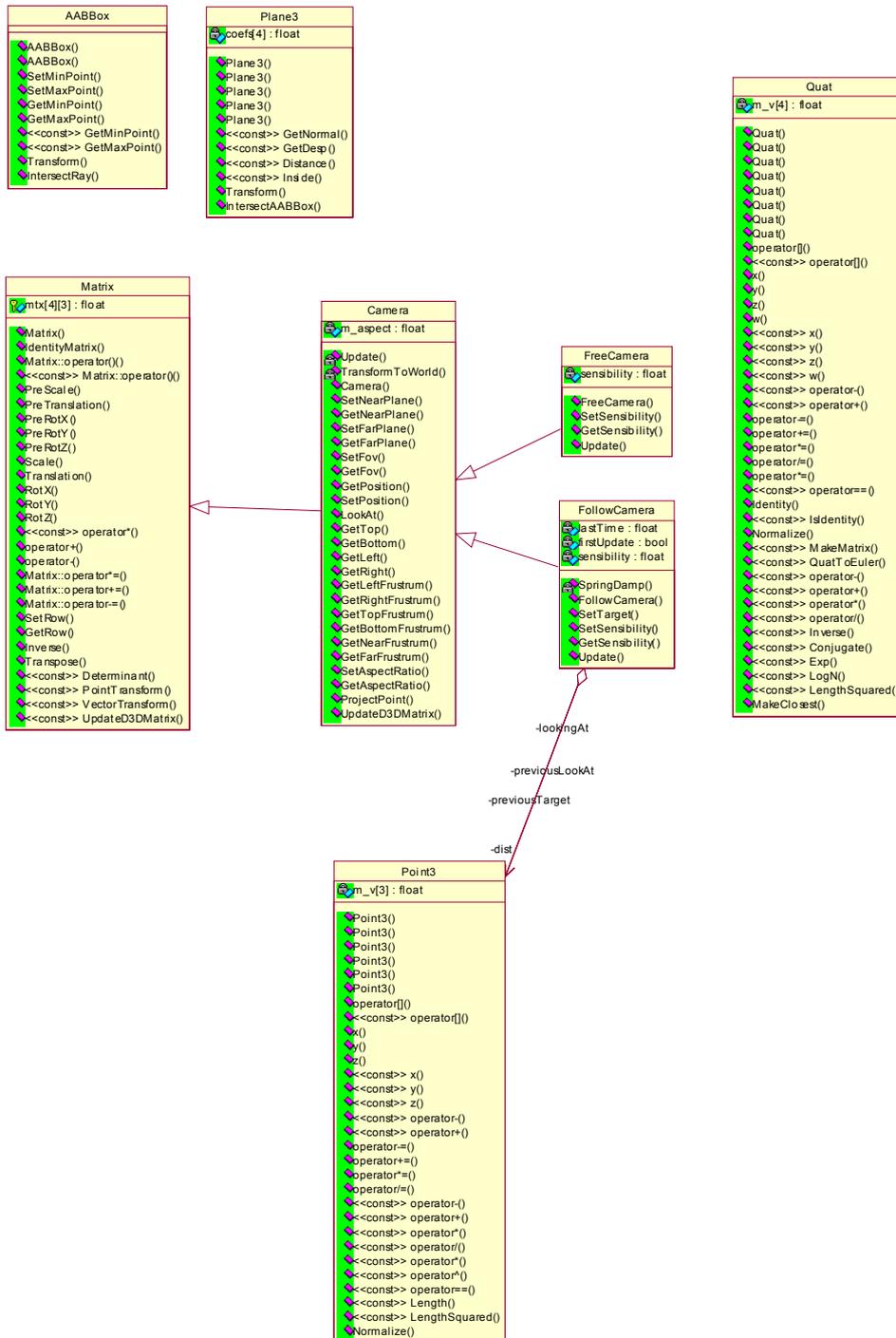
Las funciones que crean o aceptan sockets normalmente no devuelven SOCKET_ERROR, sino INVALID_SOCKET.

Esto ha sido un chequeo de errores muy basico, pero despues de SOCKET_ERROR o INVALID_SOCKET la aplicación debería llamar a int WSAGetLastError (void). WSAGetLastError devolverá un código de error que debermos interpretar.

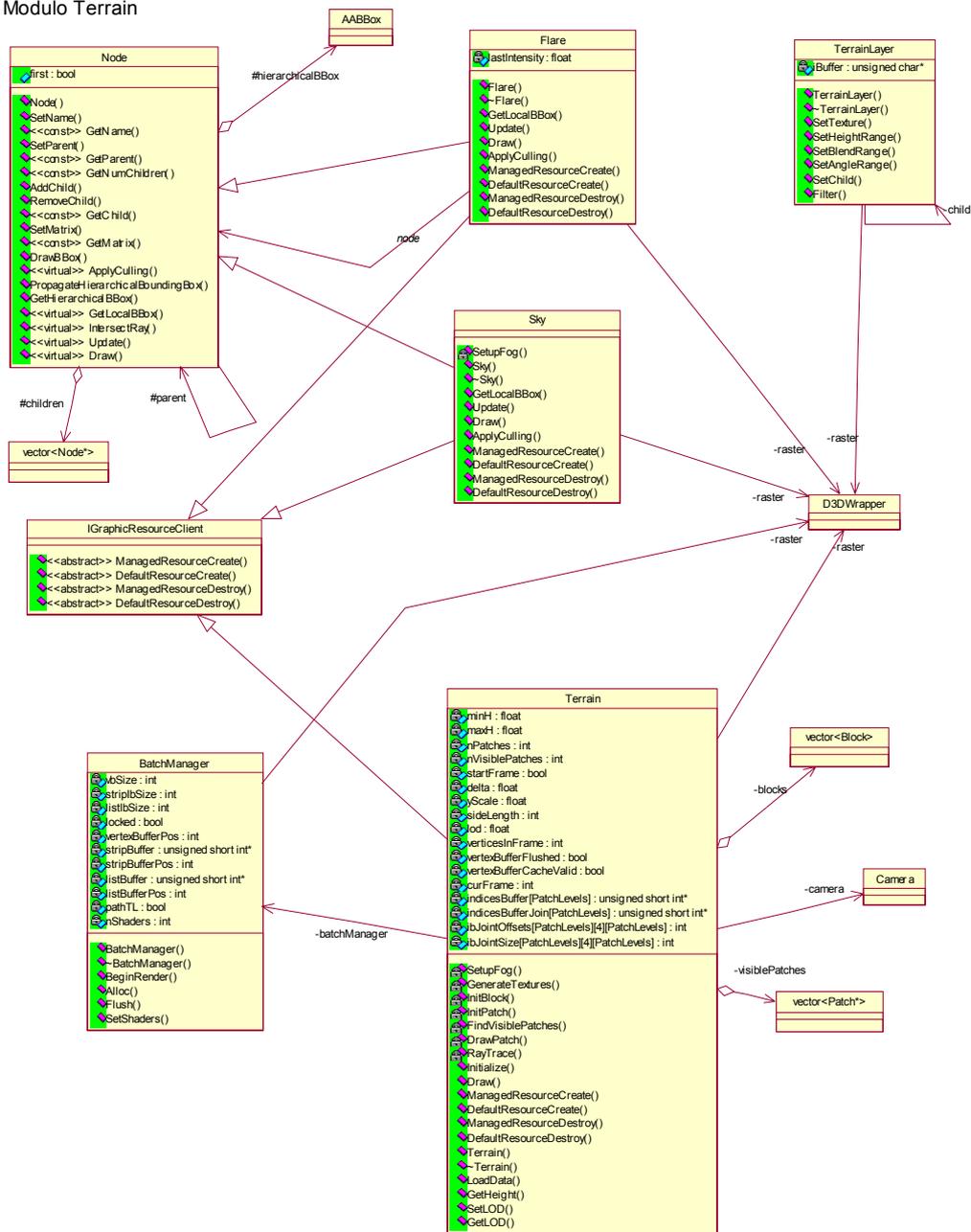
7. RELACION DE CLASES

A continuación mostramos el conjunto de clases que forman el proyecto. Para la notación de los diagramas de clases hemos empleado el lenguaje UML. Hemos dividido los esquemas por módulos.

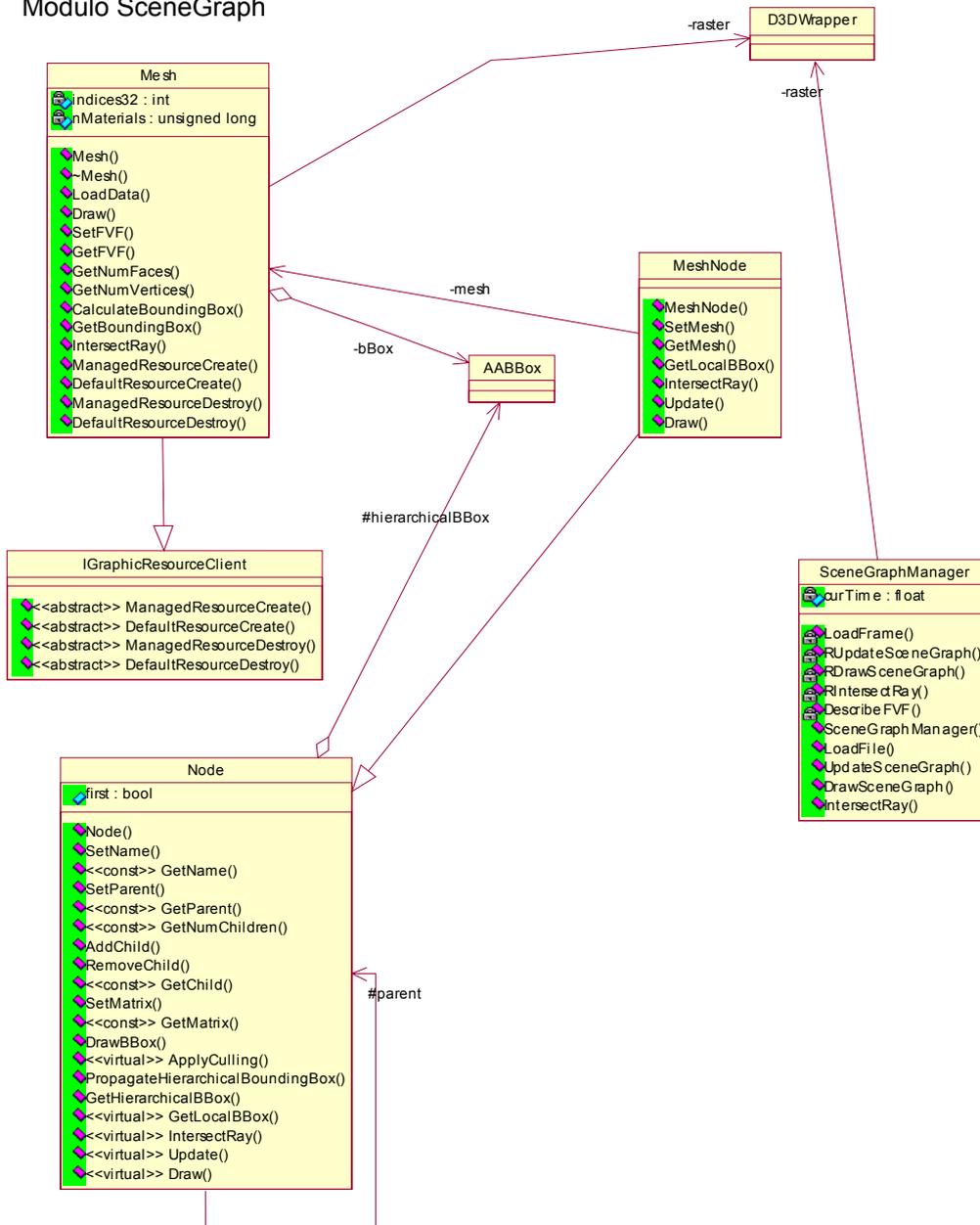
Modulo Math



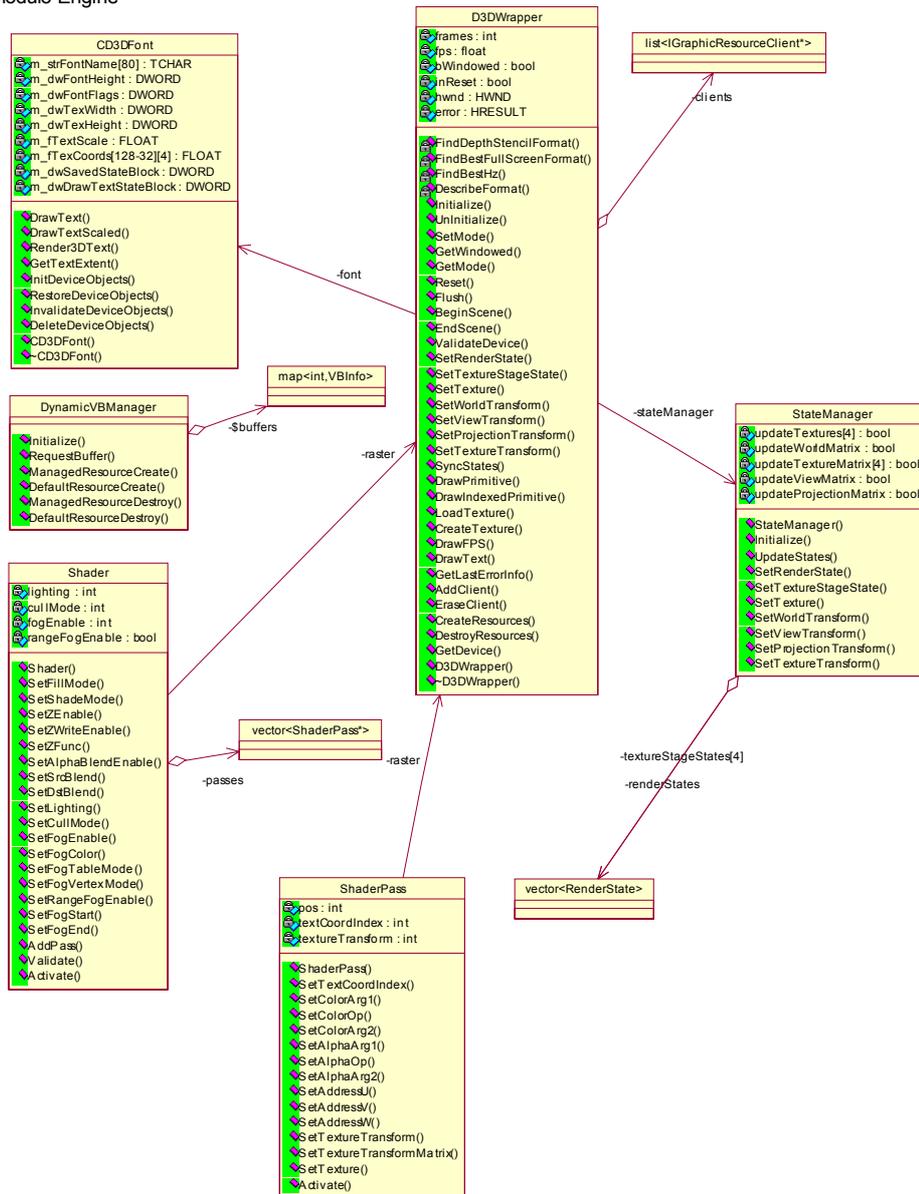
Modulo Terrain



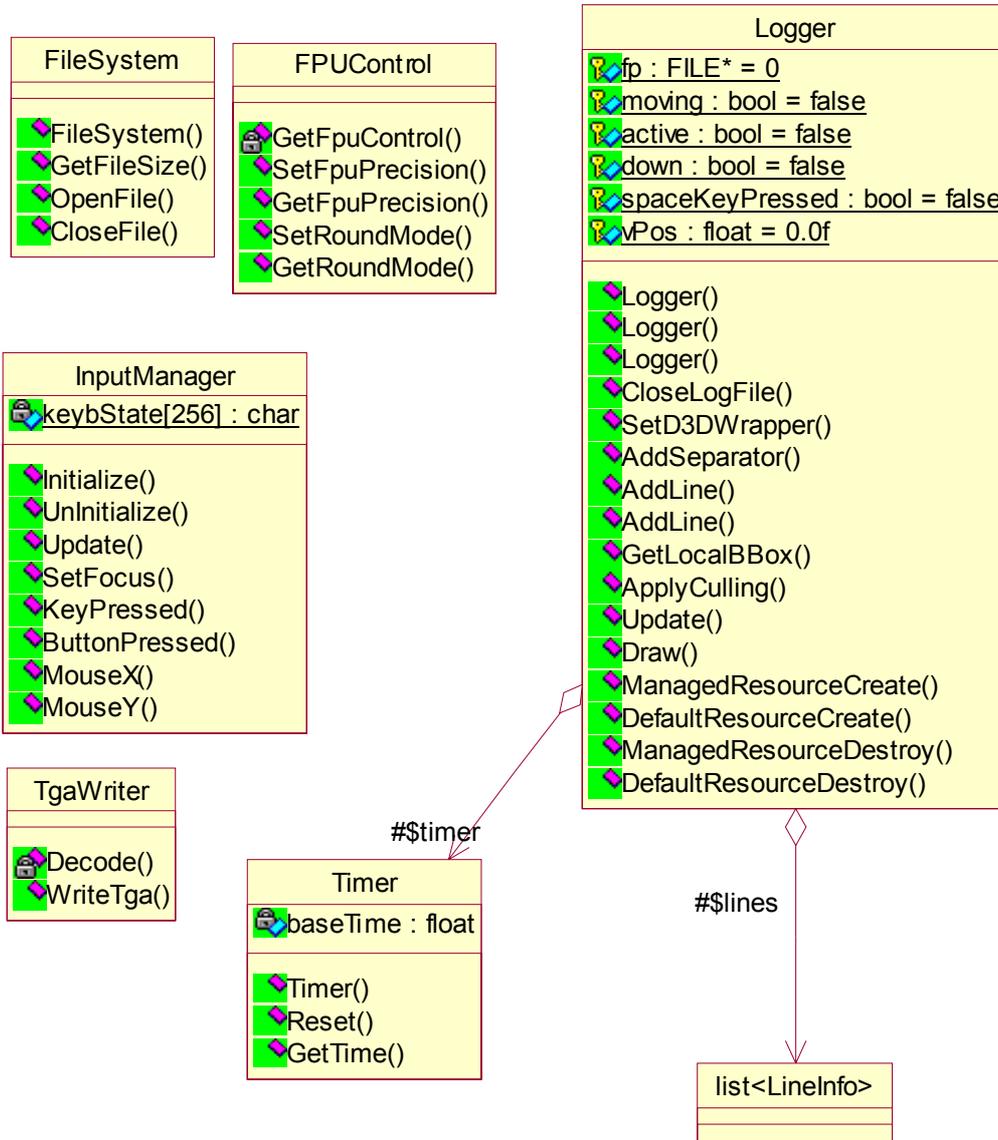
Modulo SceneGraph

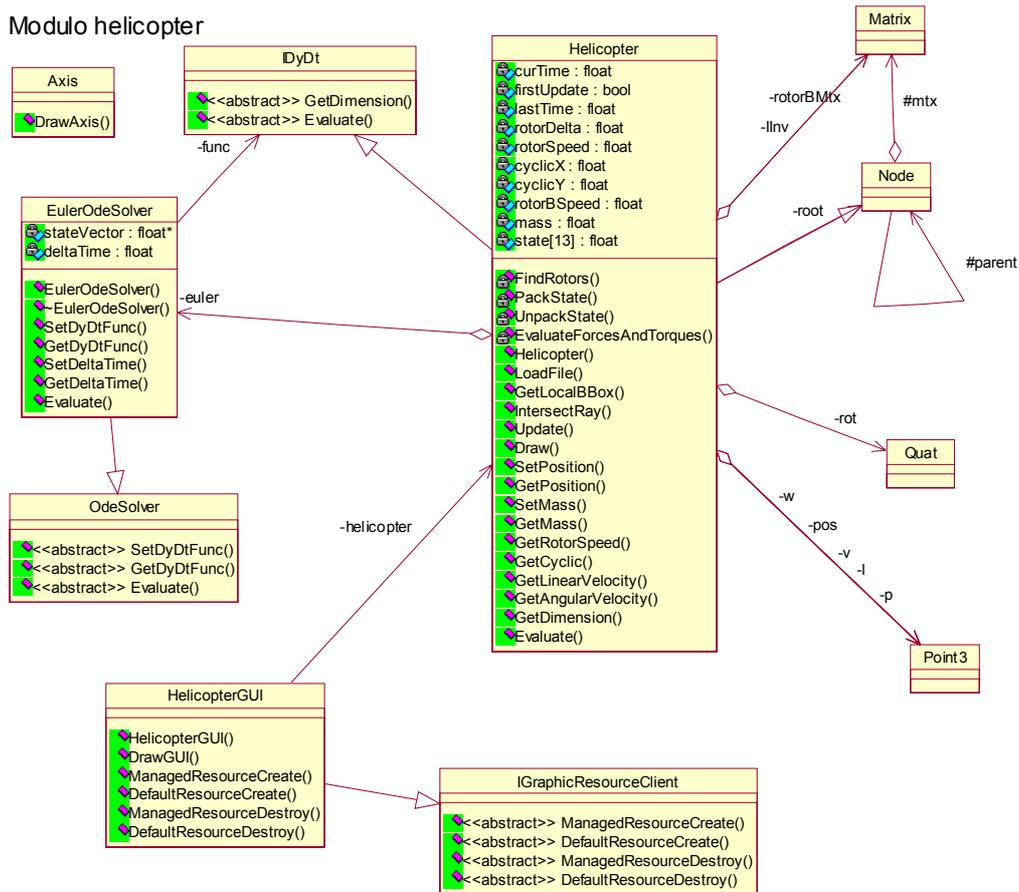


Modulo Engine



Modulo Utility





Documentación de código generada con Doc++

A continuación mostramos una documentación mas profunda del conjunto de las clases que forman el proyecto. Esta documentación se ha generado automáticamente a partir del código y sus comentarios. Con Doc++ obtenemos un conjunto de páginas webs que permiten navegar a través del código. También hemos incluido en el CD que acompaña a esta práctica el conjunto de páginas webs generadas.

8. BIBLIOGRAFÍA

- [1] ROAMing Terrain: Real-time Optimally Adapting Meshes. Mark Duchanieau, LLNL, Murray Wolinsky, LANL, David E. Sigeti, LANL, Mark C. Mille, LLNL, Charles Aldrich, LANL, Mark B. Mineev, Weinstein, LANL.
<http://www.llnl.gov/graphics/ROAM/>
- [2] Real-Time, Continuous Level of Detail Rendering of Height Fields. Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner. En los proceedings del ACM SIGGRAPH 96. Agosto 1996, pp. 109-118
<http://www.cc.gatech.edu/gvu/people/peter.lindstrom/>
- [3] Computer Graphics, Principles and Practice. Foley, van Dam, Feiner, Hughes.
- [4] Object Occlusion Culling. Tim Round. Game Programming Gems.
- [5] SurRender Umbra, Reference Manual. Timo Aila, Ville Miertinen.
- [6] Numerical Recipes in C. W.H. Press, B.P. Flannery, S.A. Teulolsky, y W.T Vetterling. Cambridge University Press Cambridge, England 1988
- [7] An Introduction to Physically Based Modeling: Differential Equation Basics. Andrew Witkin y David Baraff. Robotics Institute. Carnegie Mellon University
- [8] An Introduction to Continuum Dynamics for Computer Graphics. Michael Kass. Pixar
- [9] An Introduction to Physically Based Modeling: Rigid Body Simulation I – Unconstrained Rigid Body. David Baraff. Robotics Institute. Carnegie Mellon University.
- [10] Integrating the Equations of Rigid Body Motion. Miguel Gomez. Game Programming Gems
- [11] 3D Game Engine Design, a practical approach to Real-Time Computer Graphics. David H. Eberly. Morgan Kaufmann. 2000.
- [12] Helicopter Flight Dynamics: The Theory and Application of Flying Quantities and Simulation Modeling. G. D. Padfield. AIAA 1996
- [13] A Practical Analytic Model for Daylight. A. J. Preetham, Peter Shirley, Brian Smits. University of Utah.
- [14] Computer Graphics: principles and practice. James D. Foley. 2nd ed. c1990.
- [15] 3D Computer Graphics. Allan Watt. 2nd ed. 1996.

- [16] Real-Time Rendering. Tomas Möller, Eric Haines. 1999
- [17] Real-Time Photorealistic Terrain Lighting. Hoffman, N., y K. Mitchell. 2001
Game Developers Conference Proceedings, Marzo 2001: 357-367
- [18] An Introduction to Physically Based Modelling: Rigid Body Simulation II –
Nonpenetration Constraints.
- [19] Graphics Gems. Andrew S. Glassner
- [20] 3D Computer Graphics. Alan Watt. Third Edition Addison Wesley 2000.

A. APÉNDICES

Descripción de la primera aproximación a un objeto sobre un terreno en tres dimensiones

Objetivo

El objetivo de este programa era mostrar una primera aproximación muy sencilla de un objeto en 3D (en este caso un cubo) sobre un terreno.

Esto nos servirá como una primera aproximación práctica al futuro desarrollo de un escenario 3D con el helicóptero.

Descripción

El programa de ejemplo está desarrollado para DirectX 7, y puede ser ejecutado en un equipo que no disponga de tarjeta aceleradora 3D, puesto que puede realizar todas las operaciones necesarias por software.

Consiste en un escenario, que consta de un cubo en tres dimensiones, y un escenario también en tres dimensiones generado aleatoriamente (generamos los distintos puntos del escenario mediante un pequeño algoritmo que va generando nuevos puntos cada vez). El escenario, a pesar de ser generado así, será siempre el mismo porque siempre utilizamos la misma semilla, garantizando que el escenario sea funcional (hemos comprobado que es valido para poderlo ver en relación al cubo).

Podemos realizar distintos cambios a la posición de la cámara, y la posición del cubo, que pasamos a describir.

La cámara la podemos acercar o alejar, y rotar respecto al punto de visión hacia derecha / izquierda y arriba / abajo.

El cubo se puede acercar o alejar, y se puede rotar (siempre en torno al primer punto que se genera de él – lo que podría ser cambiado a otro, como el centro de masas) hacia la derecha / izquierda y arriba / abajo.

No existe ningún tipo de interacción entre el cubo y el escenario, en caso de que lleguen a tener puntos de intersección simplemente se verá aquello que esté delante (la malla del cubo o la del terreno, aquellas partes que sean visibles desde el ángulo de cámara).

Modo de Uso

Para poder utilizar el programa, es preciso disponer de un equipo con Windows instalado y las librerías DirectX 7 o superiores. Están no están disponibles para Windows NT 4.0 en ninguna de sus versiones.

No es preciso disponer de tarjeta gráfica aceleradora, y ha sido probado con las siguientes tarjetas:

- Windows 2000 Professional, con una ATI Rage IIc (no aceleradora)
- Windows Millenium, con una Diamond Viper 550 (Nvidia TNT)
- Windows 2000 Professional, con una Nvidia GeForce
- Windows 98 (equipo portátil del director del proyecto) donde ocurrió un error de ejecución sin saber debido a que, aunque probablemente puede ser debido a que la versión de los drivers de la controladora gráfica eran demasiado anticuados

La funcionalidad que se va a obtener será a partir del uso exclusivo del teclado:

- Flecha arriba: Movemos la camara hacia delante
- Flecha abajo: Movemos la camara hacia atrás
- Flecha derecha: Rotamos la camara en torno al punto de visión hacia la derecha
- Flecha izquierda: Rotamos la camara en torno al punto de visión hacia la izquierda
- Av. Página: Rotamos la camara en torno al punto de visión hacia abajo
- Re. Página: Rotamos la camara en torno al punto de visión hacia arriba
- 'w': Rotamos el cubo hacia abajo
- 's': Rotamos el cubo hacia arriba
- 't': Alejamos la camara
- 'g': Acercamos la camara
- 'f': Rotamos el cubo hacia la izquierda
- 'h': Rotamos el cubo hacia la derecha