A mis padres, Julia y Fernando,
que tanto han luchado por verme
llegar hasta aquí.
a Luís y Celia, a los que en la
distancia, tanto echo de menos.
A Ani por su continuo apoyo,
comprensión y ayuda. Y al cariño
y generosidad que encontré en
mi segunda familia de las palmas,

Mi más sinceros agradecimientos
a mis tutores, O.Maeso y
J. Aznárez, por su compromiso y
responsabilidad con este
proyecto. por sus consejos, ayuda
y amabilidad en el trato diario.

Contenidos

INTRODUCCIÓN

1. Antecedentes.

Las computadoras están ineludiblemente presentes en nuestras vidas. Controlan sistemas complejos tales como redes financieras, el transporte de personas, sistemas telefónicos y plantas de generación de energía. La computadora se ha convertido en un componente intrínseco de la vida moderna.

La importancia en la época actual de las computadoras y las aplicaciones informáticas en el desarrollo de la ciencia es indudable. En el campo de la ingeniería la computación ha permitido la resolución de problemas con relativa facilidad. La potencia de cálculo que dan los ordenadores permiten abarcar problemas que, antes del uso de la computación, no habían podido ni siquiera ser planteados.

La escala de los eventos simulados en las simulaciones por ordenador ha superado de lejos todo posible modelado matemático usando el tradicional lápiz y papel. Para entender la magnitud de los problemas de los que se ocupa la computación actualmente, notar que hace 10 años, una simulación de una batalla en el desierto, de una fuerza invadiendo otra, implicó el modelado de 66.239

tanques, camiones y otros vehículos en un terreno simulado alrededor de Kuwait, utilizando múltiples supercomputadoras en el *High Performance Computer Modernization Program* del Departamento de Defensa Norteamericano; un modelo de 100 millones de átomos en un modelo de deformación de materiales (2002); 2,64 millones de átomos en un modelo de fabricación compleja de proteína en todos los organismos, un ribosoma, en 2005; el proyecto *Blue Brain* en el *EPFL* (Suiza), comenzó en mayo del 2005, la creación la primera simulación por ordenador de un cerebro humano completo, a un nivel molecular.

La ciencia ha dado a las computadoras papeles principales en la simulación y resolución analítica de problemas. Sin embargo, la funciones de postprocesado, visualización, observación y análisis han quedado relegada a un papel secundario a pesar de la potencia gráfica de las máquinas actuales y de las posibilidades que dan actualmente los gráficos por ordenador.

Los gráficos realizados por ordenador (*Computer Graphics* o *CG*) es el campo de la informática en el cual, mediante el uso de computadoras, se generan imágenes sintéticas o se modifica la información visual de una imagen real.

Este campo puede ser especialmente útil en la representación de resultados de simulaciones por ordenador. El ordenador puede ocultar las matemáticas que describen un fenómeno permitiendo observarlo y familiarizarse con él antes de introducir su formulación matemática. En segundo lugar, proporciona la posibilidad de realizar medidas y modificar los parámetros que describen el fenómeno permitiendo promover la investigación por descubrimiento con el fin de desarrollar las teorías estudiadas.

De esta forma, aunque el uso de los ordenadores para la resolución de problemas es muy importante, se ha desplazado a un segundo plano el análisis de los resultados por medios informáticos. La computación abre un mundo de posibilidades que no deben infravalorarse.

Mercado de programas informáticos científicos

En la actualidad podemos encontrar en el mercado un número relativamente grande de aplicaciones para el cálculo y análisis de estructuras. Estos programas son herramientas complejas con una amplia curva de aprendizaje. Difícilmente pueden ser utilizadas por personas que no se dediquen al mundo científico, sin nociones en el campo de estudio que se realiza el análisis. Son paquetes compuestos por menús imbricados, innumerables módulos y un alto número de opciones que, si bien las dotan de gran potencia, también obligan al usuario a un amplio periodo de estudio de manuales y bibliografía sobre la aplicación.

Por lo general, estas aplicaciones centran su atención en tareas de preprocesado y cálculo, siendo estos módulos los que marcan normalmente su diseño y funcionamiento. Los cálculos se realizan normalmente de forma interna, no visible por la persona que utiliza el programa.

Estas aplicaciones también se ocupan de la representación gráfica de los resultados, sirviendo de apoyo para el análisis y el estudio por parte del científico. Esta fase de postprocesado tiene como fin mostrar de forma descriptiva los resultados, fruto de los procesos realizados en los módulos anteriores.

Enfoque del Proyecto

En este contexto se enmarca este proyecto: desarrollo de una aplicación informática para visualización y análisis de soluciones numérica en problemas dinámicos.

Las representaciones gráficas mostradas por la inmensa mayoría de estos programas se encuentran lejos de las posibilidades y logros conseguidos en otros campos de la informática, como el de la infografía y los gráficos por ordenador.

Se plantea el desarrollo de un programa que se centre en el postprocesado. Esta tarea la realizará con una filosofía más allá de la meramente descriptiva. La aplicación proyectada busca un nuevo enfoque usando las técnicas más vanguardistas e innovadoras de gráficos por ordenador y la tecnología digital para el desarrollo de potentes simulaciones científicas.

Mediante una representación más realista y una manipulación más eficiente de la información gráfica, en comparación a las tradicionales representaciones de los módulos de postprocesado existentes en el mercado, se podrá estudiar con mayor detalle los procesos que mediante representaciones más simples no puede comprobarse.

Las utilidades prácticas de este nuevo enfoque son muchas. En el campo de la sanidad, mediante una visualización cercana a la realidad es posible mejorar el diagnóstico y el control de enfermos mediante un análisis no traumático o invasivo, estudios de injertos mecánicos, etc.

En campos en los que se realiza medición de parámetros, la representación detallada de los resultados permiten el análisis en tiempo real. Por ejemplo, experimentos aerodinámicos en túneles de viento pueden ser potenciados con el uso de un postprocesado con esta filosofía de desarrollo.

Desde el punto de vista de la divulgación, la posibilidad de crear imágenes y vídeos de gran calidad aporta material que posteriormente puede incluirse en formatos como revistas, publicaciones, memorias, presentaciones, documentales, etc. También es especialmente útil desde el punto de vista del alumno, que puede interactuar con el problema, analizar los efectos de los cambios en los parámetros que rige el mismo y realizar un aprendizaje mediante prueba-error.

2. Objetivos

De acuerdo con la memoria que acompañaba a la solicitud de título de PFC, estudiado y aprobado por la Comisión de Proyectos de Fin de Carrera en octubre del 2007, se persigue como objetivo la realización de una aplicación informática que permita funciones de post-procesado de soluciones obtenidas en problemas dinámicos mediante el método de elementos finitos o elementos de contorno.

El proyecto se centrará en análisis estructural mediante mallas de elementos finitos o elementos de contorno. El post-procesado consistirá en la representación gráfica de la malla del cuerpo, estructura o dominio analizado, así como las soluciones de las variables estudiadas en el problema.

Concretamente y en base a lo anterior se plantea que la aplicación informática realizada alcance los siguientes objetivos:

- Lectura, traducción y representación gráfica en tres dimensiones de mallas de elementos mediante la lectura de un fichero de datos proporcionado.
- Lectura, traducción y representación de la solución de la/s variable/s de las que conste el problema (desplazamientos, tensiones, temperatura, etc.) y conforme a resultados matemáticos almacenados en un fichero externo.
- Representación gráfica de la evolución temporal de la soluciones numéricas del problema en cuestión.
- Operaciones de representación que permitan el análisis en profundidad de los resultados gráficos obtenidos.

Aunque el límite del tamaño de la malla con los que podrá trabajar la aplicación depende de la computadora a utilizar, se trabajará con la posibilidad de trabajar con mallas que incorporen un número de elementos muy por encima de las 5.000 caras, 10.000 nodos y 15.000 bordes con cualquier equipo de gama media que se

encuentre actualmente en el mercado. Este número posibilitará abordar proyectos de ingeniería que involucren un número considerable de grados de libertad, comparable a un problema realista de interés práctico.

Para el desarrollo del proyecto se utilizará lenguajes orientados a objetos de alto nivel como son lenguaje C++ y Java. Igualmente se trabajará para que su uso sea posible en los tres sistemas operativos más utilizados en la actualidad: Windows, Linux y Mac OS X.

3. Estructura de la Memoria

La memoria descriptiva de este proyecto está estructurada en 13 capítulos. Se comienza con un capítulo inicial (en el que se engloba este apartado) en el cual se hace una pequeña introducción sobre el problema existente, su solución y los objetivos que se pretenden cumplir.

El segundo capítulo trata de describir de forma general el uso de las computadoras en la simulación científica. A lo largo del mismo se construyen los puntos más importantes de esta disciplina. Se expone su utilidad en el mercado y las herramientas matemáticas mas utilizadas. Con mayor detalle se analiza el tema del postprocesado, tarea en la que se centra *Elesys*[1].

El capítulo 3 comienza analizando el panorama de la industria y el mercado en el que se enmarca la aplicación que sirve para tener perspectiva para extraer las necesidades, objetivos y filosofía que tiene Elesys. Así, finalmente, se analizan las alternativas generales en el desarrollo del programa y se decide la mejor solución argumentando claramente la elección.

En el cuarto capítulo se hace un breve repaso de los principios fundamentales en la programación en Maya. Se pueden leer aquí las principales características presentes en este programa.

---

[1] *Nombre que recibe la aplicación desarrollada en este proyecto*

El capítulo 5 construye los conocimientos teóricos básicos en los que se cimienta el desarrollo de Elesys. Se exponen conceptos fundamentales sin los cuales es imposible entender la filosofía de trabajo, programación y funcionamiento de nuestro programa. Se analiza la arquitectura, enfoque e ideas esenciales de Maya.

El análisis entre las dos vertientes del desarrollo para Maya se ve en el capítulo 6. Se puede encontrar las debilidades, las fortalezas y los principales aspectos del API C++ y el lenguaje MEL. Con esto, se puede decidir en cada momento cual de las dos herramientas es más conveniente usar en función de la tarea a realizar.

Con conocimientos claros sobre el desarrollo de aplicaciones en Maya, en el capítulo 5 podremos observar el diseño general de Elesys. Se expone aquí cada uno de los elementos constitutivos de la aplicación, que se verá con más detalle en secciones posteriores.

Con el capítulo 6 se comienza a explicar en detalle cómo se ha implementado cada uno de los módulos. En esta parte de la memoria se expone cómo se ha realizado la programación y cómo afronta la aplicación las tareas de verificación de archivos y la gestión de errores.

La generación de la malla a partir de ficheros externos es uno de los grandes objetivos marcados en el primer capítulo. El capítulo 9 comienza con una exposición teórica acerca de los tipos de mallas infográficas existentes. Se comenta detalladamente las mallas poligonales, elegidas para que sean las usadas por Elesys. La última parte del capítulo consta de las particularidades contempladas para las tareas de construcción de los modelos importados en los ficheros externos. Se puede ver los métodos de creación de mallas implementados en Elesys.

Los detalles en la generación de movimientos pueden verse en el capítulo 10. Se trata la animación por vértices y los retos que ésta plantea. Por último se explica

cómo Elesys ha abordado esos retos y cómo se ha desarrollado su módulo de animación para conseguir unos resultados satisfactorios.

La siguiente sección está ocupada por las tareas de edición que pueden realizarse con el fin de un mejor análisis del problema. Los detalles de cada uno de ellos y cómo se han diseñado también vienen expuesto.

En el capítulo 12 se ve todo lo relacionado con la contextualización de la escena y su renderizado final. Se habla de los artificios gráficos que pone Elesys a disposición del usuario para obtener resultados visuales muy superiores a otros módulos de postprocesado.

Por último, se termina exponiendo la interfaz de usuario. Principalmente se habla sobre los principios de diseño y filosofía que se ha seguido en la creación de la *GUI*[2].

Como es habitual en este tipo de documentos, se adjunta en las últimas páginas del mismo, la bibliografía y otros recursos utilizados para la confección de este documento.

[2] *Denominación en el campo de la informática de la interfaz gráfica de usuario, en Idioma inglés Graphical User Interface.*

Simulación por Ordenador

Desde que Kenneth Wilson, premio Nobel de Física en 1982, articuló la simulación como el tercer paradigma de la ciencia a mediados de 1980, al mismo nivel que la ciencia experimental y teórica, las simulaciones han llegado a ser instrumentos imprescindibles en muchos campos de la ciencia y la ingeniería. Las simulaciones científicas son utilizadas para probar automóviles frente a impactos antes de ser construidos, para estudiar la interacción entre un injerto de cadena y el fémur, para evaluar y restaurar puentes, para construir túneles de vientos virtuales, etc.

La simulación de procesos es una de las mayores herramientas de la ingeniería industrial. Puede definirse como la representación de un proceso o fenómeno mediante otro más simple, que permite analizar sus características.

En particular, son especialmente interesantes los sistemas de simulación numérica como el análisis de elementos finitos o elementos de contorno. Estos juegan un papel fundamental en la ingeniería por su habilidad para integrar múltiples fenómenos físicos como el flujo de un fluido, interacción entre fluido y

sólido, y comportamientos de materiales. Los sistemas de computación de análisis de elementos finitos o de contorno calculan una variedad de parámetros físicos de la simulación en el tiempo como son la posición, la velocidad, la aceleración, la tensión y la presión.

1. Simulación en el Desarrollo de Productos

Las grandes compañías del mercado han obligado en los últimos años a implantar en las empresas todas aquellas tecnologías que puedan a hacer realidad los tres grandes objetivos del diseño moderno:

- Diseñar para conseguir una fabricación a un costo competitivo.
- Diseñar, en orden, la utilización real en servicio.
- Diseñar bien al primer intento.

En este sentido la introducción del C.A.D. (*Computer Aided Design*), herramientas computacionales para actividades de diseño, está ya representando un gran avance en la etapa del diseño conceptual de nuevos productos.

Por contra, el C.A.E. (*Computer Aided Engineering*), se encuentra en una etapa mucho más primaria. El C.A.E. es el conjunto de programas informáticos que permiten analizar y simular los diseños de ingeniería realizados con el ordenador, o creados de otro modo e introducidos en el ordenador, para valorar sus características, propiedades, viabilidad y rentabilidad. Su finalidad es optimizar su desarrollo y consecuentes costos de fabricación y reducir al máximo las pruebas para la obtención del producto deseado.

Por último encontramos el C.A.M. (*Computer Aided Manufacturing* o Manufactura asistida por computadora). La manufactura asistida por computadora (CAM, de

computer aided manufacturing), implica el uso de computadores y tecnología de cómputo para ayudar en todas las fases de la manufactura de un producto.

La verdadera reducción del bucle diseño-desarrollo se produce cuando estas técnicas actúan conjuntamente. La primera para definir el producto, la segunda para simular su comportamiento en las condiciones de servicio y la tercera para definir el proceso de producción. Sólo la conjunción de estas técnicas hacen posible alcanzar los tres objetivos antes mencionados.

Actualmente estos sistemas están conectados de tal forma que ya desde la fase de diseño se puede saber el coste del producto final, controlar los stocks de componentes y materiales para su fabricación.

Hemos pasado de tener una representación de un plano en pantalla a tener un modelo virtual del que podemos obtener datos, montar en otros modelos, hacerlo adaptativo, imprimirlo y fabricarlo.

2. Simulación de Problemas Físicos Matemáticos

La gran evolución de los métodos informáticos tanto en su aspecto de hardware como software, ha permitido afrontar la resolución de complejos problemas físicos matemáticos cuya resolución analítica resultaría prácticamente imposible. De hecho muchos de estos problemas hace ya años que están planteados, sólo falta un medio adecuado para la obtención de resultados prácticos.

Así pues, la simulación intenta reproducir la realidad a partir de resolución numérica mediante ordenador de las ecuaciones matemáticas que describen dicha realidad. Por lo tanto, hay que asumir que la simulación es tan exacta como sean las ecuaciones de partida y la capacidad de los ordenadores para resolverlas.

El método de los elementos finitos y método de elementos de contorno son las técnicas de simulación más importantes y, seguramente, las más utilizadas en las

aplicaciones industriales. Las aplicaciones prácticas pueden agruparse en dos grandes familias: los problemas asociados con sistemas discretos y los problemas asociados a sistemas continuos.

En los sistemas discretos, el análisis está dividido de forma natural en elementos claramente definidos. En problemas de sistemas continuos, los sistemas no pueden ser divididos de forma natural en unidades simples, por lo que su análisis resulta mucho más complejo.

Por lo que se hace referencia al cálculo estructural, el método de elementos finitos (M.E.F.) y el método de elementos de contorno (M.E.C) pueden ser entendidos como una generalización de análisis de sistemas continuos. El principio de los métodos consiste en la reducción de un problema con infinitos grados de libertad, a un problema finito en el que intervengan un número finito de variables asociadas a ciertos puntos característicos (nodos). Las incógnitas del problema dejan de ser funciones matemáticas y pasan a ser los valores de dichas funciones en un número finito de puntos.

Así pues se supone que el comportamiento mecánico de cada parte o elemento, en los que se subdivide queda definido por un número finito de parámetros (grados de libertad) asociados a los puntos que en dicho momento se une al resto de los elementos de su entorno (nodos). Para definir el comportamiento en el interior de cada elemento, se supone que dentro del mismo todo queda perfectamente definido a partir de lo que sucede en los nodos a través de una adecuada función de interpolación.

Como puede apreciarse en lo dicho, en los dos métodos son esenciales los conceptos de "discretización" o acción de transformar la realidad de la naturaleza continua en un modelo discreto aproximado y de "interpolación", o acción de aproximar los valores de una función a partir de su conocimiento en un número

discreto de puntos. Por lo tanto el M.E.F. y el M.E.C. es un método aproximado desde múltiples perspectivas.

- Discretización.
- Interpolación.
- Utilización de métodos numéricos.

Esta presentación aproximada de la realidad en forma de un modelo numérico permite la resolución del problema. Los diversos coeficientes del modelo son automáticamente calculados por el ordenador a partir de la geometría y propiedades físicas de cada elemento. Sin embargo queda en manos del usuario decir hasta qué punto la discretización utilizada en el modelo representa adecuadamente la estructura.

La discretización correcta depende de diversos factores como son el tipo de información que se desea extraer del modelo o tipo de solicitación aplicada.

Actualmente, el método de los elementos finitos y el método de elementos de contorno ha sido generalizado hasta constituir un potente método de cálculo numérico, capaz de resolver cualquier problema de la física formulable como un sistema de ecuaciones, abarcando los problemas de la mecánica de fluidos, de la transferencia de calor, del magnetismo, etc.

A pesar de su carácter aproximado, son unas herramientas muy útiles que permiten realizar una gran cantidad de análisis de componentes y estructuras complejas, que difícilmente podrían realizarse por los métodos analíticos clásicos.

3. Estructura y Funciones de un Programa M.E.F / M.E.C.

Como puede imaginarse después de lo expuesto, un programa de elementos finitos y elementos de contorno es una pieza compleja de software en la que

confluyen numerosas operaciones. Por este motivo suelen estar divididos en subsecciones, cada una de las cuales efectúan una operación determinada.

Sin embargo, el tema no se limita al puro cálculo. La preparación de los datos y el análisis de los resultados numéricos que aparecen como producto del cálculo, son tareas arduas que actualmente se tienden a integrar en el propio software. Así pues, un paquete de cálculo consta de un preprocesador y un procesador, en el cual se incluyen todas las ayudas a la preparación de los datos y que generan los archivos de resultados, y un postprocesador que facilita el análisis e interpretación de los resultados, generalmente de forma gráfica mediante trazado de curvas, gráficos tridimensionales, tablas, etc.

El proceso de postprocesado será realizado por la aplicación desarrollada en este proyecto. Esta tarea será presentada en capítulos posteriores con gran detalle. Por ello, ahora nos centraremos en las tareas y utilidades del procesador, que veremos a continuación de forma general.

El procesador puede realizar, entre otros, los siguientes tipos de análisis:

- El análisis estático: permite la determinación de los componentes de los modos por efecto de una solicitación estática y, en una segunda fase, la determinación del estado en ciertos puntos característicos de cada elemento.

  Este tipo de análisis permite acotar la deformación del componente de estudio y localizar zonas altamente solicitadas o zonas de solicitación baja, según que el interés resida en evaluar la resistencia estática o en eliminar material.

- El análisis dinámico: puede ser de cualquiera de los tres tipos siguientes

  - Cálculo de las frecuencias y modos propios de vibración: La vibración libre de un cuerpo elástico se realiza en frecuencia y tomando formas que

le son características, denominadas frecuencias y modos propios de vibración.

El análisis de modos y frecuencias propias de vibración se realiza con el objetivo de conocer mejor el comportamiento dinámico del componente o estructura y determinar posibles áreas de conflicto, como por ejemplo la generación de resonancia.

- Cálculo de la respuesta en función del sistema: Este tipo de análisis permite determinar la respuesta vibratoria y tensional de una estructura cuando es excitada mediante una carga senoidal periódica de amplitud y frecuencia variable. De este modo es posible explorar la presencia de los diversos modos de vibración en el rango de frecuencias de interés a fin de determinar su importancia relativa.

- Cálculo y respuesta a una solicitación transitoria: En este tipo de análisis se pretende simular el efecto de una secuencia de carga real sobre la estructura, incorporando los efectos dinámicos.

- Transferencia de calor: puede abordarse problemas de conducción, convección o radiación. Los resultados son básicamente las distribuciones de temperatura y los flujos de calor.

- Mecánica de fluidos: Pueden ser problemas en el régimen laminar o turbulento, estacionario, transitorios, etc. Los resultados son básicamente las distribuciones de presión y velocidad. Si es preciso, puede incorporarse el efecto de análisis de fluidos.

- Electromagnetismo: Pueden tratarse problemas relacionados con los campos y las ondas electromagnéticas. Los resultados son básicamente los campos eléctricos y magnéticos, las distribuciones de potencial, las corrientes, los flujos magnéticos, etc.

La simulación no sólo se da en el campo de la industria, medicina o el entretenimiento. También se da en campos como la arquitectura y el entrenamiento militar. La simulación en todos estos campos tiene algo en común, la creación de un mundo irreal, pero muy real para las personas que diariamente dependen de su existencia, en campos como la calidad industrial, la medicina, el entrenamiento, los juegos, etc.

4. Postprocesado

Después de que un modelo haya sido preparado y verificado, se le hayan aplicado las condiciones de contorno y haya sido resuelto, llega el momento de observar los resultados del análisis. Esta actividad es conocida como la fase de postprocesado.

En este marco se establece el desarrollo del proyecto. La aplicación diseñada se denomina Elesys (sistema de análisis de elementos) y tendrá como cometido la realización de tareas de postprocesado a partir de datos de salida proporcionado por procesadores externos.

A continuación se expondrá de forma general y resumida las tareas que realiza Elesys para la realización de las tareas de postprocesado. En capítulos posteriores se expondrá cada una de estas tareas de forma más detallada.

El proceso debe comenzar por una verificación de los datos de entrada. Estos datos proporcionados como resultado del procesado del problema, se deben someter a un chequeo que asegure su integridad y su lógica dentro del formato establecido. Esta verificación tiene como finalidad la comprobación de la corrección matemática de los resultados. En este chequeo se busca la corrección de los datos aportados desde el punto de vista formal, es decir, se comprueban, por ejemplo, que los resultados tengan carácter numérico, que no falte ninguno

de los parámetros necesarios para la representación, la corrección en la composición de los elementos, etc.

Posteriormente, se procede a la representación de la malla, compuesta por los nodos y elementos correspondientes. Como se mencionó anteriormente, el número de nodos y elementos vendrá dado en función del grado de refinamiento buscado en las soluciones del problema. Se debe procurar que la cantidad se encuentre en un intervalo razonable. No se debe obtener una solución lejos de la solución exacta por un número de nodos pequeños y un exceso de valores interpolados, ni un modelo no manejable por el sistema de computación a causa de un número exagerado de nodos con los que trabajar.

Una vez representados los nodos y elementos, es el momento de importar las soluciones obtenidas como solución. Aquí podemos hacer una clasificación entre:

- Variables de posición de los nodos: como su nombre indica, nos aportan información sobre la posición de cada uno de los nodos a lo largo del tiempo.

- Otras variables: nos informan sobre el resto de magnitudes que intervienen en el cuerpo tales como la temperatura, presión, tensión, etc.

El carácter de esta clasificación tiene su razón de ser en los efectos que producen en la visualización. Las variables de posición nos permite dotar de movimiento al modelo. Desde el punto de vista del postprocesado, nos permite deformar y animar la malla, con lo que sus efectos son evidentes. Las otras variables no pueden verse en la visualización de forma intuitiva. Obliga al postprocesador a utilizar algún tipo de artificio que permita ver su progreso en el tiempo, como por ejemplo, la representación por escala de colores.

Es evidente que variables como la tensión, presión o temperatura, provocan de forma indirecta la deformación del modelo. Sin embargo, hay que tener en

cuenta, que la variable de posición integra todos estos efectos indirectos dando valores finales de la colocación en el espacio de cada nodo. Desde el punto de vista del postprocesador, es indiferente el origen del desplazamiento, sólo importa su valor final.

Importadas las variables en el modelo, sólo queda seleccionar la forma de visualizarlo por pantalla, así se eligen parámetros como el punto de vista de la cámara, las texturas o colores a utilizar, la iluminación de la escena, el uso de contorneado, etc.

Es aquí donde Elesys muestra sus mayores diferencia y su mayor potencia mediante el uso de técnicas infográficas. Los objetivos de este proyecto rechazan la presentación visual de los resultados por los postprocesadores genéricos existentes en el mercado. Se propone una representación utilizando los avances y tecnología desarrollada en el campo de la infografía y de los gráficos por ordenador (*Computer Graphics*). Estas técnicas y herramientas permitirán una simulación de alta calidad visual y resultados más realistas, es decir, el producto de esta simulaciones será mucho más útil para la persona que analiza el proceso simulado.

Podemos clasificar en tres la formas de representación final de resultados:

- Colores de relleno: mediante el uso de colores planos se obtienen imágenes esquemáticas donde se visualice de forma clara y concisa los resultados. No aportan mucha información pero permiten mantener la atención del observador, sin distraerlo con aspectos secundarios, gracias a su simpleza visual.

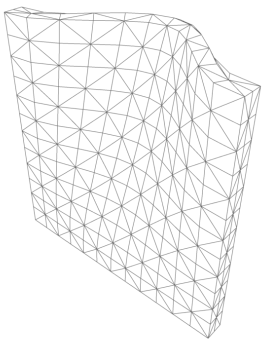


Figura 2.1. Representación con colores de relleno

- Escala de colores: posibilita visualizar de forma comparativa, variables cuyos valores no pueden comprobarse de forma visual (temperatura, presión, etc.). La escala de colores suele estar estandarizada utilizándose colores calientes, tales como el rojo o naranja, para valores altos y colores fríos, como el azul y celeste, para nodos con valores pequeños. Los valores intermedios se representan con colores verdes y amarillos.

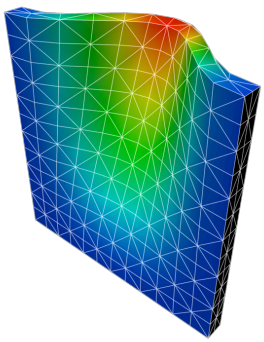


Figura 2.2. Representación en escala de colores

- Representacion fotorealista: persigue "engañar" al observador tratando de hacer creer que la imagen representada es real. Este tipo de representación es la más compleja y exige el uso de motores de renderizado de gran calidad y potencia.

Esta visualización hace que los resultados y conclusiones de la simulación sea directamente accesible por otros especialistas sin sacrificar la certeza científica. También, permite la mejor comprensión del hecho estudiado a personas no expertas en la materia. Esto hace de la representación fotorealista muy apta para conferencias, género documental y otros fines educativos.

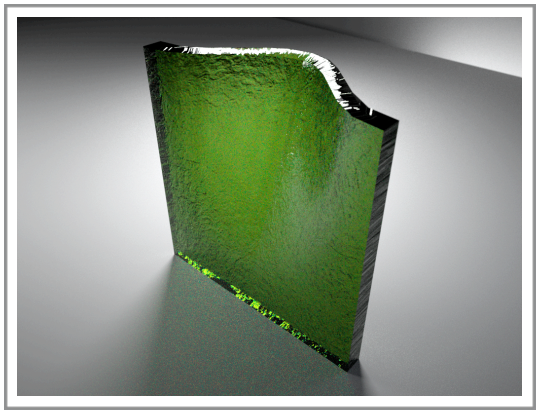


Figura 2.3. Representación fotorealista

La alta calidad visual hace de las simulaciones científicas instrumentos poderosos que serán utilizados rutinariamente en una variedad de campos inclusive la seguridad nacional, gestión de emergencia, la ciencia forense y los medios audiovisuales.

Una buena visualización llevará a mejoras de la propia simulación. Las imágenes de gran calidad revelarán rápidamente las discrepancias con datos experimentales observados con el paso de los años o registrados específicamente para realizar dicha simulación.

Capítulo 3

Estudio de Alternativas Para el Diseño de Elesys

Hemos visto el contexto en el que se va a realizar el proyecto. Analizamos la situación del mercado de los gráficos por ordenador, la industria infográfica y el entorno científico en el que la aplicación tendrá que desenvolverse. Visto de forma general el escenario en el que se engloba el proyecto estamos en disposición de analizar las posibles soluciones que podremos adoptar.

Como etapa previa al diseño de la aplicación, se pretende en este capítulo describir y estudiar las alternativas que podremos tomar para alcanzar los objetivos marcados anteriormente. Se deberá establecer un diagnóstico al problema a resolver definiendo las estrategias posibles para enfrentarse a él justificando la estrategia asumida.

Se desarrollarán las fortalezas y debilidades de cada una de las posibilidades para finalmente seleccionar el camino más idóneo. Esta decisión marcará de forma decisiva el planteamiento, diseño, desarrollo e implementación de la aplicación.

1. Planteamiento Inicial

Se puede asegurar que pocas industrias son tan dinámicas como la industria de los gráficos por ordenador. A pesar de ser un campo relativamente joven, se ha separado ya en muchas áreas distintas, que continúan avanzando con grandes mejoras y adelantos.

El principal problema que se encuentra es el alto dinamismo del mercado de la industria de gráficos por ordenador, que introduce nuevos adelantos y mejoras de forma constante, que dejan obsoletas tecnologías relativamente jóvenes.

El motor del mercado es, sin duda, el campo audiovisual. La necesidad de grandes efectos especiales, la gran aceptación que han tenido las películas de animación infográfica por parte de los espectadores y el abaratamiento de costes, hacen que los gráficos por ordenador sea un campo muy productivo e interesante para las grandes productoras cinematográficas.

El efecto secundario de la investigación promovida por un mercado económicamente tan potente, se traduce en grandes adelantos en campos interesantes para el estudio científico. Especialmente interesante es el campo de la animación. En este terreno, se busca cada vez con mayor interés la verosimilitud de la interacción de los cuerpos con el entorno cuyo realismo sólo puede obtenerse mediante herramientas matemáticas. Esto ha permitido la proliferación de paquetes que simulan con gran realismo la física de sucesos como colisiones, comportamiento de fluidos, grietas, roturas, partículas, etc.

Destaca la alta complejidad de este campo de la computación. Los gráficos por ordenador contienen una serie de conceptos específicos que obligan a adquirir ciertos conocimientos previos para poder trabajar en ellos. Estos conceptos no son algo estático, sino que se renuevan y amplían de forma continua con nuevos adelantos de la industria, que son adoptados como estándares por el todos los paquetes informáticos existentes.

2. Análisis de Requisitos

Con el fin de poder decidir cuál es la mejor de las alternativas a elegir, debemos ante todo concretar de forma clara cuáles son las características que se requieren de la aplicación a desarrollar.

Además de los objetivos concretados en la solicitud de título de PFC, estudiado y aprobado por la Comisión de Proyectos de Fin de Carrera en octubre del 2007, hay una serie de puntos que se deben de tener muy claros y presentes durante todo el proceso de creación del programa. Podemos resumirlos en:

- **Flexibilidad:** el diseño se realizará pensando en líneas futuras y posibles ampliaciones posteriores. De esta forma, en el caso de necesitarse la adaptación de funcionalidades existentes o la creación de otras nuevas en el futuro, deberá ser fácil su implementación.

- **Facilidad de uso:** A diferencia de las innumerables opciones, menús, botones, etc. que muestran los programas de postprocesado del mercado, se pretende contar con una interfaz de usuario simple y directa, sin que ello signifique una pérdida de potencia.

  Se busca que el usuario no se vea obligado a documentarse durante un largo tiempo para poder utilizar el programa con comodidad, siendo capaz de obtener resultados desde el primer momento.

- **Velocidad de proceso:** es fundamental que el proceso de simulación y representación por pantalla se realice rápidamente de forma que las modificaciones en los parámetros de la simulación provoquen que se redefina la representación de forma instantánea, sin tiempos de espera.

  Estamos ante una herramienta de análisis donde será natural el cambio continuo en los valores de los parámetros que rigen la simulación. La

rápida interacción del programa ante esos cambios es esencial si se quiere obtener una utilidad dinámica y práctica.

- **Alta calidad visual de los resultados:** es, sin duda, uno de los rasgos más distintivo de la aplicación. Se persigue unir las características de una correcta simulación científica con los últimos adelantos y avances tecnológicos en el campo de la infografía. De esta forma las imágenes resultantes tendrán una calidad muy superior a las que se podrían obtener con la mayoría de aplicaciones disponibles actualmente en el mercado.

3. Exposición de alternativas

A. Desarrollo de una aplicación completa.

La primera de las alternativas consiste en el desarrollo desde cero de una aplicación que obtenga los objetivos marcados. Esto conllevaría la creación del programa desde las capas más esenciales del mismo. El motor gráfico, núcleo de la aplicación, se debería construir a partir de las implementaciones de uno de los dos estándares gráficos predominante en el mercado. Esto son:

- *Direct3D*: es parte de DirectX, una API (*Application Programming Interface*) propiedad de Microsoft disponible tanto en los sistemas Windows de 32 y 64 bits, como para sus consolas Xbox para la programación de gráficos 3D.

  El objetivo de esta API es facilitar el manejo y trazado de entidades gráficas elementales, como líneas, polígonos y texturas, en cualquier aplicación que despliegue gráficos en 3D, así como efectuar de forma transparente transformaciones geométricas sobre dichas entidades.

- *OpenGL*: es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D. Fue desarrollada por *Silicon Graphics Inc.* (SGI) en 1992. Su nombre

viene del inglés *Open Graphics Library*, cuya traducción es biblioteca de gráficos libre, teniendo en cuenta su política de licencias.

OpenGL es una especificación, es decir, un documento que describe un conjunto de funciones y su comportamiento exacto. A partir de ella, los fabricantes de hardware crean implementaciones.

Actualmente se utiliza en campos como CAD, realidad virtual, representación científica y de información, simulación de vuelo o desarrollo de videojuegos.

B. Realización de una aplicación como plug-in

La segunda posibilidad consiste en apoyarnos en una aplicación principal desarrollando una aplicación en la modalidad plugin. Un plugin (o *plug-in*, en inglés "enchufar", también conocido como *addin*, *add-in*, *addon* o *add-on*) es una aplicación informática que interactúa con otra aplicación para aportarle una función o utilidad específica. Ésta aplicación adicional es ejecutada por la aplicación principal.

Se utilizan como una forma de expandir programas de forma modular, de manera que se puedan añadir nuevas funcionalidades sin afectar a las ya existentes ni complicar el desarrollo del programa principal.

C. Elección y Análisis de las alternativas

Después de un detallado estudio se ha optado por la solución mediante plug-in en detrimento del uso del desarrollo de una aplicación completa.

Este tipo de solución de desarrollo por plugin sigue la tendencia natural de la industria de los gráficos por ordenador cuando se persigue la realización de una tarea relativamente específica. Así, es fácil encontrar sistemas de renderizado, de simulación física, sistemas de partículas, fluidos, tejidos, etc. cuya implantación

es por el camino marcado por un plugin. A continuación se argumentará la elección tomada.

Es obvio que la principal ventaja de realizar una aplicación completa está en la posibilidad de ajustar a medida todas nuestras necesidades. Esta solución permite establecer la forma de proceder y la filosofía del programa desde sus capas más elementales.

De ambos estándares analizados, podría descartarse Direct3D puesto que no permitiría uno de los objetivos claves de nuestra aplicación. Como mencionamos anteriormente, DirectX (y más concretamente Direct3D) es un estándar cuyo funcionamiento sólo es posible en sistemas de Microsoft Windows 32 y 64 bits. El desarrollo mediante este API nos imposibilitaría su uso en otras plataformas como Linux o Mac OS X. Por tanto la elección entre ambas sería claramente a favor de OpenGL, sistema abierto compatible con las tres plataformas. Reducida la elección entre la modalidad de plug-in y OpenGL podemos analizar más concretamente otros aspectos.

El primer inconveniente del OpenGL lo encontramos en el propio contexto del mercado de los gráficos por ordenador. Como mencionamos anteriormente, estamos ante un mercado altamente dinámico, con lo que es de esperar, que lo que hoy puede suponer una solución actual, puede quedar obsoleta en poco tiempo al surgir una nueva tecnología.

Otro problema evidente está en la posibilidad de poder competir con aplicaciones desarrolladas con mayores medios económicos, tecnológicos y humanos. Es evidente, que para un proyecto de fin de carrera como el que nos ocupa, se cuenta con unos medios limitados, siendo difícil competir grandes empresas que cuentan con mayor personal, tiempo y medios para el desarrollo de sus herramientas de software.

Por contra, la principal ventaja del plugin está en la posibilidad de beneficiarnos de toda la tecnología de una aplicación principal que cuenta con años de desarrollo y una cantidad de medio dedicados que de otra forma sería imposible disponer.

La modalidad plug-in cuenta como inconveniente la dependencia que se tendría con respecto a la aplicación principal que serviría como base para nuestro programa. Se puede desarrollar la aplicación sin diseñar las capas básicas de funcionamiento pero, por contra, nos obliga a adoptar la filosofía básica de funcionamiento del programa principal.

Si bien, puede suponer un problema, cabe recordar que hasta cierto punto es un obstáculo que también encontraríamos al usar OpenGL. En este caso también se adquiriría cierta dependencia a la hora de desarrollar nuestro software. OpenGL es una API (del inglés *Application Programming Interface* - Interfaz de Programación de Aplicaciones), es decir, un conjunto de funciones y procedimientos que hay que seguir para obtener un comportamiento exacto. Por tanto, la realización de un plugin viene determinada por las reglas o normas que impone el propio sistema.

Otro problema se plantea a la hora de considerar el coste económico. La modalidad de plugin obliga a adquirir el producto principal. Esto puede suponer una sub-utilización del producto al no necesitar todas las características del programa principal, al necesitarlo sólo para tareas concretas.

También hay que tener en cuenta que la aplicación principal para la cual se programaría el plug-in debe tener versiones y poder funcionar en las tres plataformas. Es obvio que, si no existiera para algunos de los sistemas operativos mencionados, no podría utilizarse el plug-in puesto que éste no puede funcionar sin su programa base.

En el caso de elegir una aplicación base que funcionase en las tres plataformas, podría programarse el plugin con el fin de ser portado, pudiéndose utilizar de forma independiente a la plataforma elegida.

El coste de la aplicación base y la compatibilidad de las tres plataformas serán aspectos a tener en cuenta, con el fin de minimizar los inconvenientes del sistema adoptado, cuando se proceda a elegir la aplicación principal a utilizar.

4. Solución Adoptada

Finalmente se ha adoptado una solución basada en la realización de un plug-in para la aplicación Autodesk Maya. En este apartado se analizará con más detalle los motivos de la elección de este software en concreto.

Autodesk Maya permite la realización de plugins externos mediante su SDK (*Software Developer Kit*), desarrollado por la propia compañía y que se facilita junto con la aplicación Maya. El SDK nos proporciona un medio de automatizar y simplificar tareas en Maya a través del API en lenguaje C++. A través de este API se puede implantar funciones propias directamente en el núcleo de Maya.

Por ello, se ha determinado que Maya proporciona un entorno de trabajo (*framework*) donde trabajar para conseguir aquellas funciones necesarias para nuestra aplicación.

Al tener una base estable en Maya como aplicación principal, no es necesario desarrollar desde cero la aplicación. Esto nos permite concentrarnos en el diseño y programación de lo verdaderamente importante, la funcionalidad de postprocesado que se pretenden implementar.

Por otro lado, la elección tomada nos proporciona gran flexibilidad. Aunque la API C++ de Maya da unas reglas a la hora de proceder en la realización de plugin, estas son totalmente independiente de su núcleo de proceso. Por ello, no

estamos obligados a tener en cuenta el funcionamiento interno sino sólo las normas de la API.

Además, esta solución nos permite interactuar a su vez, con otros plugins realizados por terceras partes. Esto abre un gran campo a la hora de aumentar la utilidad de nuestra aplicación. Pongamos como ejemplo un plugin que simule el comportamiento de fluidos. Unido a nuestro programa nos permitirá conocer cómo se comporta un fluido que está en contacto con el cuerpo o estructura que estemos estudiando.

Esta API incluye soporte nativo para OpenGL y DirectX. Así Maya integra ambos estándares sin renunciar a ninguno y pudiendo acceder a toda la potencia de sus sombreadores por hardware.

Maya está disponible para las tres plataformas más utilizadas hoy en día en el mundo de los ordenadores como son: Windows, Linux y Mac OS X. Gracias a esto, la aplicación desarrollada podrá utilizarse en estas plataformas permitiendo al usuario seguir trabajando con el sistema operativo elegido sin necesidad de preocuparse de incompatibilidades y correcto funcionamiento de la aplicación.

El precio de la aplicación se ve reducido hasta la gratuidad para fines educativos y no comerciales. Puesto que Elesys tendrá esta finalidad, hace que el obstáculo del coste económico, uno de los principales inconveniente del desarrollo por plugin, se elimine.

Elección entre API C++ o MEL

Para realizar plugins en Maya que extienda su funcionalidad existen dos herramientas muy diferenciadas: MEL y la API C++

De las dos opciones para programar en Maya, es necesario decidir en cada momento, cual de ellas utilizar. Al escoger qué interfaz de programación se va a usar, es importante sopesar ambas a la luz de las necesidades particulares. La elección final puede depender de factores externos tales como facilidad de lectura del código o requisitos específicos de velocidad.

Generalmente MEL proporciona toda las funciones que un desarrollador medio necesita crear en un programa. Permite acceso a muchas de las funciones de Maya sin la necesidad de utilizar C++. Cuando la interfaz de programación usada es C++ normalmente es por una función específica que no puede ser encontrada en MEL.

Es importante entender que la elección de una no excluye automáticamente a la otra. La interfaz C++ no es un sobreconjunto de la interfaz MEL; es decir, que la

interfaz C++ no contiene todo lo que ofrece MEL y más. Hay algunas características que puede encontrarse en MEL que no están disponibles en C++ y viceversa. Como tal, algunos de los problemas pueden ser resueltos sólo mediante la combinación de ambas.

Sólo con una buena comprensión de ambas interfaces de programación, se podrá reconocer cuando una será mejor que la otra para una parte dada y de esa forma crear la mejor mezcla. Esta base de conocimiento sobre ambas herramientas será cubierta en este capítulo.

1. MEL (Maya Embedded Language)

MEL es un acrónimo de Maya Embedded Language. Es un lenguaje de programación a medida diseñado para trabajar específicamente en Maya. Es un lenguaje script usado para simplificar tareas en Autodesk Maya.

MEL ofrece un método de acelerar tareas complicadas y repetitivas, así como permitir a los usuarios crear funciones y procedimientos complejos. Debido a su estructura y sintaxis más sencilla, es más fácil y mucho más accesible que la programación en C++.

Una de las ventajas de MEL es que es un lenguaje interpretado. Mientras los típicos lenguajes de programación requieren que sean compilados y posteriormente linkado a su código fuente, un lenguaje interpretado puede ser ejecutado inmediatamente. Esta habilidad de ejecutar inmediatamente sus instrucciones significa que MEL es especialmente bueno para un rápido prototipado.

Con MEL puede diseñar de forma más leíble e implementar nuevas ideas, puesto que el proceso de compilación y linkado no es necesario. De hecho, MEL puede ser completamente escrito, depurado, y probado dentro de Maya. No hay necesidad de utilizar compiladores ni programas externos de depuración.

Que MEL sea un lenguaje interpretado tiene una desventaja: puede ejecutarse mucho más lento que su equivalente en C++. El código fuente de un programa en C++ es compilado para engendrar las instrucciones nativas verdaderas de la máquina, por lo que corre muy rápidamente. Un lenguaje interpretado tiene el código fuente que debe ser interpretado a la vez que se ejecuta el script. Cuando Maya encuentra una instrucción MEL, debe interpretarla y finalmente debe convertirla a la instrucción nativa de la máquina. Maya realiza esta conversión de forma eficiente y de la forma más rápida posible, sin embargo en muchas ocasiones MEL se encuentra muy por detrás de C++ en términos de velocidad. Una vez dicho esto, hay muchos casos en los que la ventaja de crear y ejecutar un programa MEL rápidamente es más importante que la configuración, complejidad, y costos de compilación de un programa en C++.

2. API C++ de Maya

Maya puede ser programada usando el lenguaje de programación estándar C++. Esta forma es, con diferencia, la manera más poderosa de extender Maya. Usando C++ puede crear plugins nativos que funcionen de forma parecida al resto del paquete.

El acceso de programación se da a través del API de C++ (*Application Programming Interface*). Esta interfaz consiste en una serie de librerías de clases en C++. Para crear un plugin simplemente hay que escribir un programa en C++ que utilice una extensión de las clases básicas de Maya.

El proceso de aprendizaje de la programación con el API de C++ conlleva el aprendizaje de algunas clases y conocer cómo se utilizan. Las clases están diseñadas de forma consistente, por lo que una vez aprendidas algunas de las clases más simples, estos conocimientos ayudarán a aprender algunas de las clases más avanzadas. Aunque el número de clases podría en principio desanimar, un plugin de Maya normal usará un número pequeño de ellas.

Generalmente, sólo se usan alrededor de una tercera parte de las clases disponibles. Algunas de las clases más extrañas es raro que sean usadas.

3. Facilidad De Uso

La elección de la interfaz con la que programar puede ser determinada simplemente por el conocimiento en programación que se posea. Mientras MEL tiene una notable similitud con el lenguaje C, es lo suficientemente diferente para hacerlo fácilmente accesible a desarrolladores menos experimentados. La eliminación de algunos elementos como punteros, localización y deslocalización de memoria, hacen que MEL sea de lejos más fácil de usar y comprender. Mientras MEL no proporciona el acceso de bajo nivel ofrecido por C, raramente hay necesidad de ello cuando programe en Maya. También, al no tener esta posibilidad previene las causas más comunes de cierres inesperados de programas e inestabilidad general.

Los scripts MEL, al estar más cerca al pseudocódigo que otros lenguajes, tienden a ser más fáciles de leer. MEL también es menos restrictivo pues no impone tantos chequeos de tipos de datos. Esto significa que se puede escribir scripts rápidamente sin tener que preocuparse por los tipos de variables que hay en ellos. Esto pueden ser una bendición o una maldición, puesto que puede dar como resultado errores sutiles que posteriormente puedan ser difíciles de encontrar. Afortunadamente se puede elegir mejorar el tipo de verificación mediante la declaración explícita del tipo de variable cuando sea definida.

Con una mayor experiencia en C++, se podría optar por utilizar el API C++. La jerarquía de las clases de Maya en C++ proporciona acceso a una gran parte de las funciones de Maya. Sin embargo, es inevitable tener que recurrir a realizar algo de programación en MEL, incluso aunque, a priori, se piense que todo puede ser escrito en C++. Una buena comprensión del lenguaje C++ será la

ayuda más acertada en el aprendizaje de MEL, pues su sintaxis es muy similar a C.

Siempre que la programación del plugin requiera complejas estructuras de datos, será mas probable que se necesite usar C++. MEL contiene un conjunto limitado de tipos variables diferentes y no le permite definir sus propias estructuras de datos.

4. Funcionalidad Específica

Es fácil asumir que C++ es más complejo que MEL y ello debería darnos mayor nivel de acceso y control. Esta afirmación es parcialmente cierta. Usando C++ se puede crear un plugin y escribir software que permita un mayor nivel de integración con el núcleo de Maya. Sin embargo es erróneo asumir que el API C++ es un sobreconjunto de las funciones de MEL. De hecho ambas interfaces son complementarias.

Hay ciertas funciones a las que se puede acceder desde MEL que no se podrá tener en C++ y viceversa. Así que al realizar ciertas tareas no habrá posibilidad de elección y se deberá de utilizar una combinación de ambas. Es bastante común ver un plugin de C++ con muchas llamadas a órdenes MEL.

El API de C++ proporciona ciertas características que simplemente es imposible de replicar en MEL. Sólo mediante el API de C++ se puede crear nodos propios que permite crear herramientas a medidas como deformadores, formas, localizadores, manipuladores, etcétera. Tampoco MEL permite crear comandos a medida, aunque se pueden crear procedimientos MEL que actúen de forma similar a ciertas órdenes.

Sin tomar en consideración la interfaz de programación que se decida para desarrollar el plugin, cuando se comience a tratar asuntos de la interfaz gráfica,

se tendrá que utilizar MEL como único medio de controlar la interfaz. Afortunadamente es posible ejecutar órdenes MEL desde la API de C++.

5. Portabilidad Entre Plataforma

Dado un extenso desarrollo de Maya en múltiples plataformas, es importante decidir si la portabilidad es un factor importante en el desarrollo de Elesys. Si se eligiera desarrollar para una sola plataforma, entonces este tema será menos importante. Sin embargo, con el uso cada vez más extendido de varias plataformas tanto para el procesamiento de datos como para el render, se considera más prudente considerar desde estas etapas, previas al diseño, las consecuencias potenciales de la decisión.

Puesto que MEL es un lenguaje en scripts, ha sido diseñado para ser portable fácilmente a otras plataformas. MEL contiene una pequeña dependencia de la plataforma. De hecho casi todas órdenes funcionan sin la consideración de en qué plataforma en particular están funcionando. Esto significa que se puede utilizar un script de MEL en múltiples plataformas de manera normalmente más fácil puesto que tiene funciones con una menor dependencia. Puesto que MEL es utilizado para todas las funciones de la interfaz gráfica, no habrá que preocuparse por las especificaciones de los distintos sistemas de ventanas. Utilizando MEL puede desarrollar la interfaz en una plataforma y saber que aparecerá similarmente en las otras.

El desarrollo en varias plataforma de software en C++ es notoriamente más difícil por muchas razones. Los problemas empiezan cuando los distintos compiladores de C++ tienen sus propias incompatibilidades. También están los grados en los que varían los convenios del lenguaje ANSI C. No todos los compiladores utilizan las mismas bibliotecas standard de C. Usando sólo las clases de Maya en C++, las bibliotecas con pequeña dependencia de otras

librerías externas y características avanzadas del lenguaje debería ser fácil el desarrollo en varías plataformas.

Atendiendo a principios de flexibilidad y con el fin de no desaprovechar el uso de Elesys en la plataforma elegida por el usuario, se considera un punto clave este punto. Será importante a la hora de la escritura del código en C++ y el uso de librerías, cuidar la compatibilidad con las tres plataformas más extendidas hoy día en el mercado: Windows, Linux y Mac OS X.

6. Velocidad

Puesto que MEL es un lenguaje interpretado, es probable que sea más lento que C ++. Esto no es siempre así, sin embargo, la velocidad depende en gran parte de la complejidad del programa.

Para ciertas tareas, la diferencia de velocidad no es lo suficientemente significativa como para justificar el esfuerzo extra de escribirlo en C++. Sin embargo, en los procesos de Elesys, que generalmente serán operaciones complejas en las que el tiempo sea realmente un aspecto crítico, no hay duda de que el programa en C++ correrá más rápido. De hecho la ganancia de velocidad puede ser de diez veces mayor. Esto es un factor importante cuando la geometría utilizada sea grande y compleja en una escena. Hay que considerar que el esfuerzo extra de programación en C++ puede cosechar grandes recompensas en interactividad y productividad general del usuario.

7. Decisiones finales

El desarrollo de Elesys cuenta con una serie de peculiaridades y objetivos que marcan en qué medida debe elegirse la interfaz de la API C++ o la interfaz por lenguaje de script MEL.

Para obtener los resultados deseados, Elesys está obligada a realizar un gran número de cálculos complejos. La magnitud de esta actividad computacional exige la solución más rápida si no se quiere que la velocidad y eficiencia de la aplicación disminuya excesivamente. Una construcción basada en su mayoría en la API C++, dota claramente de mayor velocidad al paquete y permitirá una gran velocidad de proceso.

Uno de los módulos claves es el importador de datos. La información primaria que contará Elesys para realizar sus procedimientos, no vienen de la propia aplicación, sino que debe ser obtenida del exterior. Este proceso de importación es clave en el plugin. Sin la realización correcta de esta tarea sería imposible alcanzar los objetivos posteriores y tampoco podría garantizarse la obtención de resultados correctos.

Las tareas de verificación de la consistencia de datos y normalización del formato establecido son realizadas de forma más segura, rápida y eficiente por el lenguaje C++. En la actualidad resulta imposible obtener herramientas similares en MEL a las que proporciona C++ con librerías que dotan al programa con buffer de datos complejos, expresiones regulares, etc. Por tanto, en estas tareas deberá también optarse por la API C++.

La necesidad de obtener la aplicación para múltiples plataformas, con el fin de no obligar al usuario a trabajar en un sistema operativo concreto, hace que lenguaje MEL sea interesante para algunas tareas. Especialmente útil resulta en el desarrollo de la GUI de Elesys, en el que nos asegura la consistencia y el idéntico funcionamiento en las tres plataformas en las que se desarrolla el programa.

Por último, necesitaremos comandos y funciones estándar de Maya para establecer parámetros y realizar tareas ya existentes en el paquete de Autodesk y que son especialmente útiles para los fines de este proyecto. Mediante MEL se

puede acceder a estas funciones ya implementadas en el núcleo de Maya de forma más clara, directa y fácil.

De esta manera, se utilizará la API de C++ para la realización de la mayor parte de Elesys. Esto no significa que se relegue a MEL, que será usado para el desarrollo de la interfaz gráfica de usuario y para la realización de llamadas a funciones estándar de Maya. Estas llamadas se realizarán mediante las posibilidades que permite la API de Maya de llamar a sentencias MEL.

Diseño General

Establecidas las bases teóricas, el enfoque y objetivos en el desarrollo de Elesys, es el momento de plantear el diseño de la aplicación. Resulta esencial un correcto diseño de la aplicación de forma general.

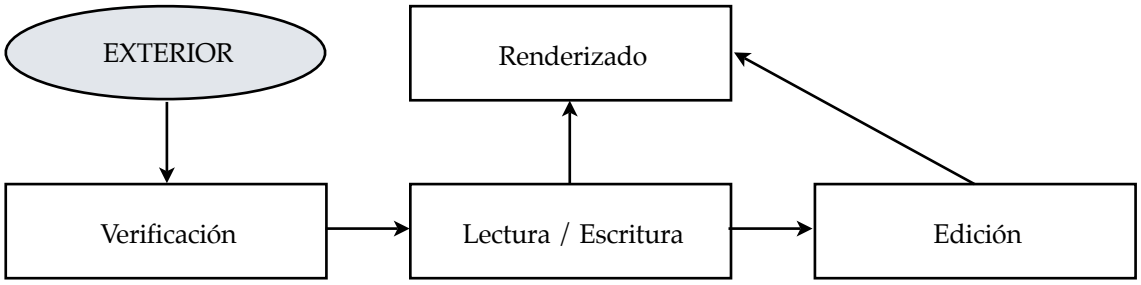


Figura 5.1. Esquema general de módulos de Elesys

En este capítulo se determinará la propuesta de trabajo de acuerdo a pautas y a procedimientos sistemáticos. Este punto facilitará el desarrollo de etapas posteriores y clarificará la finalidad de cada uno de los módulos de los que se compone la aplicación.

1. Modularidad

El diseño de la aplicación se realiza de forma modular, separando claramente cada tarea. Este enfoque permite una definición más clara del problema, abordando la situación de forma más asequible al separar la gran complejidad de la aplicación en dificultades más pequeñas.

El primer paso es definir los módulos que compondrá la aplicación. Estos deben establecerse conforme a los procesos a realizar y objetivos a alcanzar. De esta forma, y después de un largo análisis y estudio, los módulos se han establecidos de la siguiente manera:

1. Módulo de verificación. Se ocupa de comprobar la corrección del fichero aportado por la aplicación externa a Elesys. Deberá establecer si el formato seguido por el fichero externo es correcto y se atiene a la normalización realizada.

   Como tarea adicional, en caso de detección de algún error, deberá indicar la situación y proporcionar la información más concreta posible al usuario que le permita identificar el problema subyacente en su fichero o módulo de exportación.

2. Módulo de lectura/escritura. Su tarea es la de leer y escribir el archivo verificado correctamente en la fase anterior, adaptándolo al formato utilizado por Maya.

3. Módulo de edición. Tendrá que realizar tareas de modificación sobre los datos ya importados que permitan un mejor análisis de los resultados del problema.

4. Módulo de renderizado. Su finalidad es obtener representaciones de alta calidad del problema estudiado en formato de imagen o video.

Es importante atender bien a las conexiones entre los distintos módulos debiéndose identificar correctamente los lazos de unión entre cada parte de la aplicación. La mejora, modificación o la implementación de nuevas funciones en uno de los módulos no debe obligar a modificar los módulos adyacentes con los que se comunique. Esto haría entrar en un efecto rizado que haría muy difícil el desarrollo.

La figura 5.1. muestra de forma esquemática la disposición relativa entre los distintos módulos. Se observa que las tareas de verificación y lectura/escritura son altamente dependientes. La relación entre ambas está muy definida y es, igualmente, la menos dificultosa.

La tarea de verificación no realiza ninguna modificación sobre el fichero original, de forma que la entrada y la salida del módulo de verificación es el propio fichero original proveniente del exterior.

Por su parte, el módulo de lectura/escritura obtiene los datos existentes en el fichero inicial. Posteriormente, deberá procesarse, resultando la información en el formato que indica o necesita la API y MEL, para su posterior uso en Maya. Por tanto, no podemos decir que la salida de este módulo esté claramente identificada, ya que depende de los posteriores procesos que se realizarán en los módulos de edición y renderizado a los que está conectado.

Es evidente que los datos de entrada que necesitará el módulo de renderizado, para la realización de sus tareas, diferirán de los requeridos por el módulo de edición. Se debe atender cuáles de estos datos deben ser proporcionados por el fichero de datos originales y cuales deben ser establecidos por el usuario durante el uso de Elesys.

Los datos de salida del módulo de edición deberán construirse de la misma forma que se hace en el módulo de lectura/escritura cuando su comunicación es con el módulo de render. Debe cuidarse que, en el proceso de edición, se alteren

de forma adecuada los datos necesarios por el módulo de render para que la representación final sea correcta.

Concretemos con el siguiente ejemplo. Supongamos que tenemos una malla con un movimiento determinado, resultado de importar en Elesys un fichero de elementos de contorno y un fichero de movimiento según una frecuencia determinada. A continuación, para una mejor visualización, se quiere exagerar el movimiento aumentando la amplitud del movimiento al doble. Realizamos por último una representación en el que los puntos que superen un cierto valor de amplitud, se coloreen en rojo con el fin de identificar qué nodos superan un límite establecido de rotura.

Se ve cómo desde el punto de la visualización del movimiento la edición es útil. El movimiento se ve de forma más clara al exagerarse sus efectos y puede ayudar a detectar polos, máximos, etc. Sin embargo, desde el punto de vista de la detección de posibles roturas es preferible los datos originales sin modificar. El coloreado de la figura no tendrá utilidad si no se puede relacionar fehacientemente con una magnitud real.

Con ello se comprueba cómo la conexión entre proceso de edición y de representación debe realizarse con precaución, estableciendo si se deben proporcionar los datos originales o modificados según el caso a representar.

2. Generalidades del Módulo de Verificación.

Como comentamos anteriormente, el módulo de verificación se ocupa de asegurar la integridad y corrección del fichero a importar, identificando e informando de los errores encontrados si los hubiera.

La entrada de este módulo está claramente definida, así como el fichero resultante de la exportación del software destinado al preprocesado y procesado

del problema a analizar. Con respecto a estos ficheros se debe establecer dos aspectos:

- La información que podrá contener el fichero.
- Formato en el que deberá estar escrito dicho contenido para que sea leíble y entendible por Elesys.

Por tanto, de forma previa al propio diseño del módulo, se tiene que establecer una serie de condiciones y normalizaciones de la información a recibir. Estamos ante un proceso muy sistemático, que obliga a establecer una serie de reglas estrictas. Sin embargo, aunque este paso obligado permitirá un correcto funcionamiento del módulo encargado de la identificación de errores, también tiene sus riesgos. Al restringir la información a recibir, también limitamos la flexibilidad de la aplicación. Debe cuidarse mucho la determinación de estas reglas a imponer con el fin de no tener que modificarse posteriormente. En su diseño debe contemplarse con perspectiva la aplicación completa, los resultados que se quiere obtener, con el fin de no omitir alguna información necesaria para futuros procesos.

Por otro lado, en caso de error, la identificación de problema resulta vital. Sin una detección e información eficiente de los errores encontrados, resultará muy difícil subsanarlos y permitir que la aplicación pueda comenzar a trabajar con los datos.

Hay que tener en cuenta que el desarrollo de los módulos exportadores, que permitirán la conexión entre un software determinado y Elesys, será realizado por terceras partes. Si se quiere facilitar el uso y garantizar el éxito de Elesys como módulo de postprocesado, es vital facilitar al máximo las tareas de desarrollo de exportadores de datos al formato establecido. Una detección de errores fiable y clara permitirá al desarrollador externo del exportador depurarlo correctamente.

Para este módulo se utilizará procesos de control de excepciones y un uso importante de expresiones regulares mediante la libreria en C++ REGEX. Los detalles y documentación técnica se ampliará en capítulos posteriores de esta memoria.

3. Generalidades del Módulo de Lectura/Escritura.

El módulo de lectura/escritura parte de dos premisas:

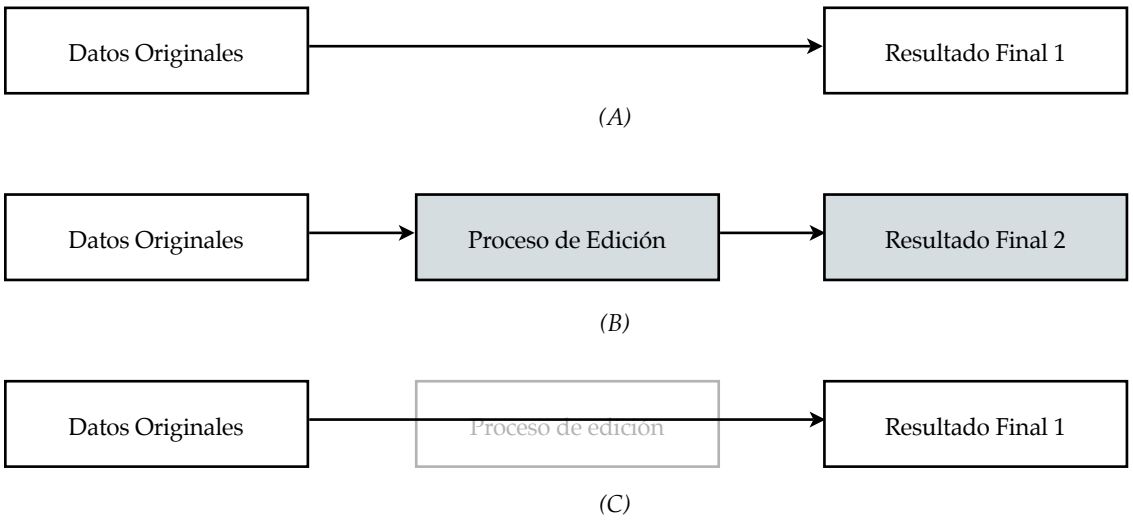


Figura 5.2. Esquema de edición no destructiva

- Se recibe el fichero original dado por el exportador de la aplicación externa. Se entiende que el módulo de verificación no ha realizado ningún proceso de edición sobre los datos.
- Este fichero original se encuentra dentro de los límites establecidos en la normalización de ficheros. La información que contiene está correctamente estructurada y no cuenta con errores de formato.

Estas condiciones, que asume el módulo como cierta, es responsabilidad del módulo que le transfiere la información de entrada, es decir, del módulo de verificación comentado anteriormente.

El primer paso a realizar es la lectura del fichero. Ésta debe ser rápida y eficaz. Para ello debe minimizarse el número de veces que se recorrerá el archivo en su lectura. Para este proceso se utilizará las jerarquías de clases de alto nivel de C++: istream, ostream, iostream y las demás subclases que se derivan de ellas.

El resultado de estas lecturas, deben almacenarse adecuadamente en cadenas de datos. Estas cadenas de datos deben cumplir, en lo posible, dos condiciones: mínimo tamaño y máxima velocidad en la recuperación de los datos en el futuro.

La escritura de los datos deben realizarse en ficheros con el formato adecuado que entiende Maya. Siempre que se pueda, se inclinará por la escritura en formato binario por las ventajas de velocidad de lectura y menor tamaño de fichero resultante.

4. Generalidades del Módulo de Edición.

La característica más particular de este módulo es su objetivo de modificar los datos originales aportados por el módulo externo. Esta particularidad esencial es lo que le hace especialmente complejo.

Además de ser capaz de realizar las tareas propias de edición de la forma más eficiente posible, se debe diseñar estas tareas de forma que realicen una edición no destructivas del modelo. Es decir, el módulo edición debe editar la malla sin realizar modificaciones en los datos originales, de forma que pueda deshacerse las modificaciones en el momento que se quiera.

La propia filosofía de malla, que define la escena con el DG compuesto por los correspondientes nodos y conexiones, permitirán programar esta edición no destructiva de forma correcta. Las ediciones, como las demás tareas de Elesys, se realizarán mediante la creación de nodos con sus respectivos atributos. La supresión de estos nodos de edición deben provocar la restauración del modelo original.

Esta filosofía de trabajo permite realizar las tareas de edición sin preocupación por parte del usuario a destruir datos importantes que puedan ser necesarios en el futuro. De forma contraria, el usuario se vería obligado a reiniciar el proceso de postprocesado cada vez que necesitara recuperar algún dato original destruido por la edición aplicada.

En la figura 5.2. puede verse la filosofía de la edición no destructiva. De forma esquemática, se puede ver la aplicación de una tarea de edición y la posterior supresión de esta sin pérdida de datos originales.

El apartado (A) muestra el DG original, cuyo valor de salida final es el "Resultado Final 1". Si le aplicamos el nodo "Proceso de Edición", como se ve en el apartado (B), con sus respectivos atributos y función de computación, editará la salida de los "Datos Originales" y se tiene el "Resultado Final 2", diferente del "Resultado Final 1". La edición no destructiva permite que, al borrar el nodo Edición, se vuelva a obtener el "Resultado Final 1" como se tenía en un primer momento en el apartado (A).

Los procesos de edición que se plantean que Elesys debe realizar son:

- Ocultación de elementos.
- Suavizado de superficies.
- Simetrías.
- Antisimetrías.

Los detalles de estas ediciones se comentarán en capítulos posteriores.

5. Generalidades del Módulo de Representación.

En el último módulo de Elesys, renderizado, se aprovechará el motor gráfico disponible en Maya para la representación final de imágenes y vídeos. Maya cuenta con múltiples motores de render. Algunos se pueden encontrar integrados en el paquete básico del programa y otros pueden adquirirse como plugins desarrollados por empresas externas a Autodesk.

La primera decisión importante a tomar a la hora de afrontar el diseño de este módulo es qué motor de render utilizar e implementar en Elesys. La elección, teniendo en cuenta la gran cantidad y calidad de plugins de representación existentes en el mercado, no será fácil.

Se debe tener en cuenta los objetivos esenciales a cumplir por el motor de render utilizado. Estos pueden resumirse en:

- Calidad de renderizado.
- Bajo coste económico.
- Rapidez de render.
- Posibilidad de programación mediante SDK.
- Flexibilidad de configuración.
- Posibilidad de coloreado por vértices.

En el desarrollo de este módulo se incidirá en tareas de creación de materiales, creación de sombreadores (*shaders*) y determinación de parámetros de representación

Verificación De Ficheros

Una de las cualidades de Elesys es que es capaz de aceptar mallas de terceras aplicaciones siempre que se las proporcionen de forma adecuada. Las ventajas de este sistema son numerosas. En primer lugar, dota de enorme flexibilidad al usuario que no tiene restringido el uso de una herramienta concreta para tareas de preprocesado, pudiendo usar la aplicación que considere más adecuada. Por otro lado, posibilita la representación gráfica de los resultados de cualquier problema dinámico sea cual sea las variables, parámetros y contexto del problema dinámico.

Sin embargo, este sistema también constituye el primer gran problema con el que nos encontramos en la realización del proyecto. El usuario tiene como único requisito dar la malla y los resultados de su problema en un formato normalizado. En caso contrario, la aplicación no podrá trabajar con ellos y/o no podrá llegar a resultados correctos.

Si bien la construcción de estos ficheros normalizados constituyen un problema, puede resolverse fácilmente con la construcción de exportadores de ficheros que

conviertan el formato de origen al formato final establecido. La posibilidad de crear exportadores hace que se conserve la modularidad de la aplicación, tal y como se propuso desde un principio, quedando intacta la filosofía original del proyecto.

La tarea de exportar e importar los datos no constituyen, por sí solo, la única dificultad a superar en este campo. Es necesario también, realizar una verificación de la información, asegurando que los datos sean coherentes y se ajusten al formato y contenido necesario por Elesys para operar. Una correcta comprobación evitará bloqueos y cierres inesperados, resultados inconsistentes, errores inesperados, etc.

1. Reglas Generales de Formato

Para que Elesys reconozca y lea correctamente el fichero del mallado y movimiento, se deben seguir unas normas.

En cuanto a la forma de representar la información establecemos que:

- El fichero debe estar en texto sin formato, también denominado como texto plano (*plain text*). Se busca así la simplicidad, evitando la información destinada a generar formatos (negrita, subrayado, cursivas, tamaños, etc.), que, a efectos de la aplicación, no aporta ninguna utilidad.
- En cuanto a la codificación del texto se utilizará sistema ASCII (acrónimo de *American Standard Code for Information Interchange*), más concretamente *Unicode UTF8.* Este sistema es un código de caracteres basado en el alfabeto latino. El código ASCII utiliza 7 bits para representar los caracteres. Casi todos los sistemas informáticos actuales utilizan el código ASCII o una extensión compatible para representar textos y para el control de dispositivos que manejan texto.

Así se consigue que el contenido pueda ser visible y entendible por el usuario usando cualquier editor de texto. Igualmente permite la edición en el caso de que existan pequeños errores o que se quiera realizar pequeños cambios sin necesidad de generar de nuevo el archivo.

A continuación se comentan más detalles acerca del código ASCII.

Código ASCII

El código ASCII reserva los primeros 32 códigos (numerados del 0 al 31 en decimal) para caracteres de control: códigos no pensados originalmente para representar información imprimible, sino para controlar dispositivos (como impresoras) que usaban ASCII. Por ejemplo, el carácter 10 representa la función "nueva línea" (*line feed*), que hace que una impresora avance el papel.

La siguiente tabla muestra los códigos de control del código ASCII:

Binario	Decimal	Hex	Abreviatura	Nombre/Significado
0000 0000	0	00	NUL	Carácter Nulo
0000 0001	1	01	SOH	Inicio de Encabezado
0000 0010	2	02	STX	Inicio de Texto
0000 0011	3	03	ETX	Fin de Texto
0000 0100	4	04	EOT	Fin de Transmisión
0000 0101	5	05	ENQ	Enquiry
0000 0110	6	06	ACK	Acknowledgement
0000 0111	7	07	BEL	Timbre
0000 1000	8	08	BS	Retroceso
0000 1001	9	09	HT	Tabulación horizontal
0000 1010	10	0A	LF	Line feed
0000 1011	11	0B	VT	Tabulación Vertical

Binario	Decimal	Hex	Abreviatura	Nombre/Significado
0000 1100	12	0C	FF	Form feed
0000 1101	13	0D	CR	Carriage return
0000 1110	14	0E	SOH	Shift Out
0000 1111	15	0F	SI	Shift In
0001 0000	16	10	DLE	Data Link Escape
0001 0001	17	11	DC1	Device Control 1
0001 0010	18	12	DC2	Device Control 2
0001 0011	19	13	DC3	Device Control 3
0001 0100	20	14	DC4	Device Control 4
0001 0101	21	15	NAK	Negative Acknowledgement
0001 0110	22	16	SYN	Synchronous Idle
0001 0111	23	17	ETB	End of Trans. Block
0001 1000	24	18	CAN	Cancel
0001 1001	25	19	EM	End of Medium
0001 1010	26	1A	SUB	Substitute
0001 1011	27	1B	ESC	Escape
0001 1100	28	1C	FS	File Separator
0001 1101	29	1D	GS	Group Separator
0001 1110	30	1E	RS	Record Separator
0001 1111	31	1F	US	Unit Separator
0111 1111	127	7F	DEL	Delete

En cuanto a los caracteres imprimibles ASCII, el carácter espacio, designa al espacio entre palabras, y se produce normalmente al pulsar la barra espaciadora de un teclado. Los códigos del 33 al 126 se conocen como caracteres imprimibles, y representan letras, dígitos, signos de puntuación y varios símbolos.

A continuación se muestra una tabla con aquellos caracteres imprimibles:

Binario	Dec	Hex	Signo
0010 0000	32	20	espacio
0010 0001	33	21	!
0010 0010	34	22	“
0010 0011	35	23	#
0010 0100	36	24	$
0010 0101	37	25	%
0010 0110	38	26	&
0010 0111	39	27	’
0010 1000	40	28	(
0010 1001	41	29	)
0010 1010	42	2A	*
0010 1011	43	2B	+
0010 1100	44	2C	,
0010 1101	45	2D	-
0010 1110	46	2E	.
0010 1111	47	2F	/
0011 0000	48	30	0

Binario	Dec	Hex	Signo
0101 0000	80	50	P
0101 0001	81	51	Q
0101 0010	82	52	R
0101 0011	83	53	S
0101 0100	84	54	T
0101 0101	85	55	U
0101 0110	86	56	V
0101 0111	87	57	W
0101 1000	88	58	X
0101 1001	89	59	Y
0101 1010	90	5A	Z
0101 1011	91	5B	[
0101 1100	92	5C	\
0101 1101	93	5D	]
0101 1110	94	5E	^
0101 1111	95	5F	_
0110 0000	96	60	`

Binario	Dec	Hex	Signo
0011 0001	49	31	1
0011 0010	50	32	2
0011 0011	51	33	3
0011 0100	52	34	4
0011 0101	53	35	5
0011 0110	54	36	6
0011 0111	55	37	7
0011 1000	56	38	8
0011 1001	57	39	9
0011 1010	58	3A	:
0011 1011	59	3B	;
0011 1100	60	3C	<
0011 1101	61	3D	=
0011 1110	62	3E	>
0011 1111	63	3F	?
0100 0000	64	40	@
0100 0001	65	41	A
0100 0010	66	42	B

Binario	Dec	Hex	Signo
0110 0001	97	61	a
0110 0010	98	62	b
0110 0011	99	63	c
0110 0100	100	64	d
0110 0101	101	65	e
0110 0110	102	66	f
0110 0111	103	67	g
0110 1000	104	68	h
0110 1001	105	69	i
0110 1010	106	6A	j
0110 1011	107	6B	k
0110 1100	108	6C	l
0110 1101	109	6D	m
0110 1110	110	6E	n
0110 1111	111	6F	o
0111 0000	112	70	p
0111 0001	113	71	q
0111 0010	114	72	r

Binario	Dec	Hex	Signo
0100 0011	67	43	C
0100 0100	68	44	D
0100 0101	69	45	E
0100 0110	70	46	F
0100 0111	71	47	G
0100 1000	72	48	H
0100 1001	73	49	I
0100 1010	74	4A	J
0100 1011	75	4B	K
0100 1100	76	4C	L
0100 1101	77	4D	M
0100 1110	78	4E	N
0100 1111	79	4F	O

Binario	Dec	Hex	Signo
0111 0011	115	73	s
0111 0100	116	74	t
0111 0101	117	75	u
0111 0110	118	76	v
0111 0111	119	77	w
0111 1000	120	78	x
0111 1001	121	79	y
0111 1010	122	7A	z
0111 1011	123	7B	{
0111 1100	124	7C	|
0111 1101	125	7D	}
0111 1110	126	7E	~

2. Reglas del Fichero Malla

Si atendemos ahora al contenido del fichero, podemos dividir el fichero en dos partes:

1. ***Información de los nodos***: en esta primera parte se indicará la información relativa a cada uno de los nodos. Encontraremos:

- Una primera línea con un número entero N_1 que indicará el número de nodos a representar en la malla.

- Una serie de N_1 líneas con cuatro columnas tal que:

    1. La primera columna estará formada por un número entero, que será la identificación del nodo para diferenciarlo unívocamente.

    2. Las segunda, tercera y cuarta columnas que serán tres números en coma flotante con cinco decimales, que representarán las coordenadas cartesianas x, y, z del nodo.

2. ***Información de los elementos***: tras el último nodo, en la siguiente línea, encontraremos:

- Un número entero N_2 que indicará el número de elementos de los que esta formado la malla.

- Una serie de N_2 líneas con 8 u 11 columnas, según el caso, que representarán:

    1. Un número identificativo, en formato entero, para diferenciar unívocamente a cada elemento.

    2. Un número entero que representa el número de nodos que forman el elemento. Elesys acepta solamente dos tipos de elementos, de 6 y de 9 nodos.

    3. Una serie de 6 o 9 columnas de números enteros que representan los números identificativos de los nodos que forman dicho elemento.

Por último, destacar que, se utilizará el carácter '.' para representar el punto decimal y que la separación de cada columna se hará mediante el carácter '|'.

A continuación, se puede ver el contenido de un fichero mallado de un cubo formado por seis elementos:

```
45|
10|1.000000|0.000000|0.000000|
11|1.000000|0.5000000|0.000000|
12|1.000000|1.000000|0.000000|
13|1.000000|0.000000|0.5000000|
14|1.000000|0.5000000|0.5000000|
15|1.000000|1.000000|0.5000000|
16|1.000000|0.000000|1.000000|
17|1.000000|0.5000000|1.000000|
18|1.000000|1.000000|1.000000|
20|0.000000|1.000000|0.000000|
21|0.5000000|1.000000|0.000000|
22|1.000000|1.000000|0.000000|
23|0.000000|1.000000|0.5000000|
24|0.5000000|1.000000|0.5000000|
25|1.000000|1.000000|0.5000000|
26|0.000000|1.000000|1.000000|
27|0.5000000|1.000000|1.000000|
28|1.000000|1.000000|1.000000|
30|0.000000|0.000000|0.000000|
31|0.000000|0.5000000|0.000000|
```

```
32|0.000000|1.000000|0.000000|

33|0.000000|0.000000|0.5000000|

34|0.000000|0.5000000|0.5000000|

35|0.000000|1.000000|0.5000000|

36|0.000000|0.000000|1.000000|

37|0.000000|0.5000000|1.000000|

38|0.000000|1.000000|1.000000|

40|0.000000|0.000000|1.000000|

41|0.5000000|0.000000|1.000000|

42|1.000000|0.000000|1.000000|

43|0.000000|0.5000000|1.000000|

44|0.5000000|0.5000000|1.000000|

45|1.000000|0.5000000|1.000000|

46|0.000000|1.000000|1.000000|

47|0.5000000|1.000000|1.000000|

48|1.000000|1.000000|1.000000|

50|0.000000|0.000000|0.000000|

51|0.5000000|0.000000|0.000000|

52|1.000000|0.000000|0.000000|

53|0.000000|0.5000000|0.000000|

54|0.5000000|0.5000000|0.000000|

55|1.000000|0.5000000|0.000000|

56|0.000000|1.000000|0.000000|
```

```
57|0.5000000|1.000000|0.000000|

58|1.000000|1.000000|0.000000|

5|

1|9|10|11|12|15|18|17|16|13|14|

2|9|20|21|22|25|28|27|26|23|24|

3|9|30|31|32|35|38|37|36|33|34|

4|9|40|41|42|45|48|47|46|43|44|

5|9|50|51|52|55|58|57|56|53|54|
```

2. Reglas del Fichero Movimiento

Elesys acepta dos tipos de datos de movimiento. El primer tipo es utilizado para la representación del movimiento de una malla de acuerdo con una frecuencia determinada. El segundo se usa para la representación de la evolución temporal de una malla ante una acción exterior.

Cada uno de estos dos enfoques necesitan una información distinta para que Elesys pueda postprocesar el problema. Por tanto, es lógico pensar, que el contenido del fichero de movimiento en cada uno de los dos casos será distinto. Así, las reglas y la normalización del fichero de movimiento será diferente para cada uno de los dos casos.

Las particularidades de cada tipo de problema y solución serán comentados en el capítulo 10, referente a la generación del movimiento en el mallado.

Fichero de Variable en Frecuencia

El fichero de una variable en frecuencia estará compuesto por los datos necesarios para la obtención de la magnitud de dicha variable en cualquier

instante de tiempo. Con esta información, Elesys obtienen posteriormente los valores necesarios para su representación en el tiempo.

Así, se establece el siguiente formato:

- Una primera línea, con la cadena de caracteres '*frecuencia*' , que servirá para indicar a Elesys que el tipo de fichero de movimiento que está recibiendo es de frecuencia.

- Una segunda línea, con un número en coma flotante, que indicará la frecuencia del movimiento en radianes por segundo.

- Una tercera línea, con un número entero N_3 , que indicará el número de líneas a leer, correspondiente a las variable de cada nodo necesarias para representar el problema.

- Por último, una serie de N_3 líneas con diez columnas cada una, tal que:

    1. La primera columna estará formada por un número entero, que será la identificación del nodo para diferenciarlo unívocamente.

    2. Las tres siguientes columnas indicaran la posición del nodo mediante tres números enteros o flotantes con las coordenadas cartesianas x, y, z.

    3. Según sea una variable escalar o vectorial tendremos:

        3.1. De la quinta a la décima columna, serán seis números en coma flotante, que representarán la parte real y parte imaginaría de la variable en el eje x, y, z: U_{ReX}, U_{ImX}, U_{ReY}, U_{ImY}, U_{ReZ}, U_{ImZ}.

        3.2. De la quinta a la décima columna, serán seis números en coma flotante, que representarán la parte real y parte imaginaría de

la variable escalar y que se repiten en las 6 columnas: U_{Re}, U_{Im}, U_{Re}, U_{Im}, U_{Re}, U_{Im}.

Por último, destacar que, de la misma forma que para el fichero del mallado, se utilizará el carácter '.' para representar el punto decimal y que la separación de cada columna se hará mediante el carácter '|'.

Fichero de Evolución Temporal de la Variable

El fichero de movimiento de evolución temporal se compondrá de los datos de cada nodo en cada uno de los fotogramas o *frames* de la animación. El valor de estas variable no responden, a diferencia del caso anterior, al resultado de una operación matemática que permita conocer el valor en cada instante de forma fácil.

Por ello, se debe exigir al usuario que aporte a Elesys los valores finales para la variable a representar. Estos valores se obtendrán como soluciones después de una correcta tarea de preprocesado y cálculo del problema.

Con estas premisas, se establece para los ficheros de evolución temporal el siguiente formato:

- Una primera línea con la cadena de caracteres '*temporal*' que servirá para indicar a Elesys que el tipo de fichero que está recibiendo es de tipo evolución temporal.

- Una segunda línea con un entero N_1 que indicará el número de nodos de la malla.

- Una tercera línea con un número entero N_2 que indicará el número de fotogramas que se tienen que representar.

- Por último, una serie de ($N_1 \cdot N_3 + 1$) líneas con una serie de columnas tal que:

    2. La primera columna estará formada por un número entero, que será la identificación del nodo para diferenciarlo unívocamente.

    3. Una serie de columnas, según sea la variable escalar o vectorial:

        3.1. Variable vectorial: tendremos tres columnas más, con números enteros o flotantes, con los valores de la variable en las coordenadas x, y, z en dicho nodo para ese frame.

        3.2. Variable escalar: habrá igualmente tres columnas pero con el valor escalar correspondiente repetido tres veces.

Clarifiquemos lo expuesto con el siguiente ejemplo, en el que se muestra el contenido de un fichero de evolución temporal de una variable vectorial:

```
temporal|
10|
240|
1 | 3.00 | 1.23 | 4.32 |
2 | 1.63 | 5.21 | 2.85 |
3 | 3.02 | 0.12 | 0.05 |
4 | 0.05 | 1.15 | 0.64 |
5 | 0.00 | 1.77 | 1.18 |
6 | 1.10 | 0.46 | 1.35 |
7 | 1.34 | 1.22 | 2.19 |
8 | 3.03 | 2.82 | 2.65 |
```

```
9 | 0.04 | 1.46 | 0.64 |

10 | 1.45 | 1.23 | 2.32 |

11 | 2.90 | 1.27 | 4.30 |

12 | 1.60 | 5.01 | 2.72 |

13 | 2.95 | 0.03 | 0.08 |

{…}
```

Para una variable escalar el ejemplo sería:

```
temporal|

8|

160|

1 | 3.00 |

2 | 1.63 |

3 | 3.02 |

4 | 0.05 |

5 | 0.00 |

6 | 1.10 |

7 | 1.34 |

8 | 3.03 |

9 | 2.90 |

10 | 1.60 |

11 | 2.95 |

{…}
```

3. Gestión de errores.

Definidas las reglas generales para ficheros de malla y movimiento que podrá importar Elesys, se abordará a continuación la detección de errores en los procesos del programa.

Desde los inicios de los lenguajes de programación, la gestión de errores ha sido uno de los asuntos más difíciles de abordar. La fuente de errores es muy amplia: un incorrecto comportamiento del usuario, un fichero que no existe, una operación no permitida o fallos en la interacción con un periférico, etc.

Estas circunstancias pueden provocar desastres en la ejecución del programa que suponen la finalización de la aplicación de forma descontrolada, dejando ficheros abiertos, provocando pérdidas de datos, etc. Para evitar estos problemas y dotar a Elesys de una estructura robusta frente a hechos no previstos, se ha utilizado un sofisticado control de errores.

4. Errores en Maya

Chequeo

Maya proporciona mecanismos de reportes de errores a través del uso de la clase **MStatus**. Esta clase define los posibles estados para el resultado de una operación dada. Cuando la operación falla, el **MStatus** cambia al estado apropiado.

Casi todas las funciones de las clases de Maya tienen un puntero opcional a un objeto **MStatus**. Si se utiliza un puntero a un objeto **MStatus**, posteriormente Maya indica el resultado de la función llamada del objeto.

El siguiente ejemplo muestra el uso general del chequeo de errores para una función genérica de Maya:

```
MStatus stat;
```

```
funcion(&stat);
if(!stat)
    MGlobal::displayError( “Unable to get dag name” );
```

Para realizar un correcto chequeo de errores, es importante determinar si la función es exitosa o falla. Para hacer esto, se le pasa un puntero del objeto **MStatus** a la función en cuestión. El resultado de la llamada a esta función es almacenada en el objeto **stat**. Su chequeo determina el resultado del proceso realizado por la función. El contenido del objeto **stat** indica si se ha hecho la tarea de forma correcta o si, por el contrario, se ha producido algún error.

El ejemplo anterior muestra la metodología de chequeo de errores más común usada en plugins. Existe una variedad de formas de chequear el resultado de un objeto **MStatus**. Además del `if(!stat)` usado previamente, se puede usar la función `error()`.

```
if(stat.error())
    … // error
```

Desafortunadamente, el mecanismo de reporte de errores de Maya usado pone mucho énfasis en que el desarrollador sea vigilante en el chequeo de errores. Así, para el desarrollo de Elesys, se verifica el resultado de todas las funciones llamadas en Maya. A pesar de que es una tarea ardua, se entendió que es extremadamente importante mantener un consistente chequeo de errores.

Así, si se tiene una serie de llamadas a funciones que no se chequean ante llamadas erróneas es muy difícil, en caso de que se produzca algún fallo, encontrar en qué punto se produjo el problema. Los distintos *bugs* y errores en tiempo de ejecución son resueltos mucho más rápidos si se chequea cada pieza de código.

Reporte

La clase **MStatus** proporciona algunas funciones adicionales para el reporte de errores que se utilizan en Elesys. La función `errorString()` devuelve un *string* correspondiente al código del error que tiene lugar. La función `perror()` permite que imprima un mensaje de error a el correspondiente flujo `stderr`. Un ejemplo de uso de esta función se muestra a continuación:

```
if( stat.error() )
    stat.perror( MString("Unable to get name. Error: ") +
stat.errorString() );
```

El *Command Feedback* de Maya, es un espacio situado por defecto en la esquina inferior derecha del visor de Maya, que sirve de medio de comunicación con el usuario del programa.

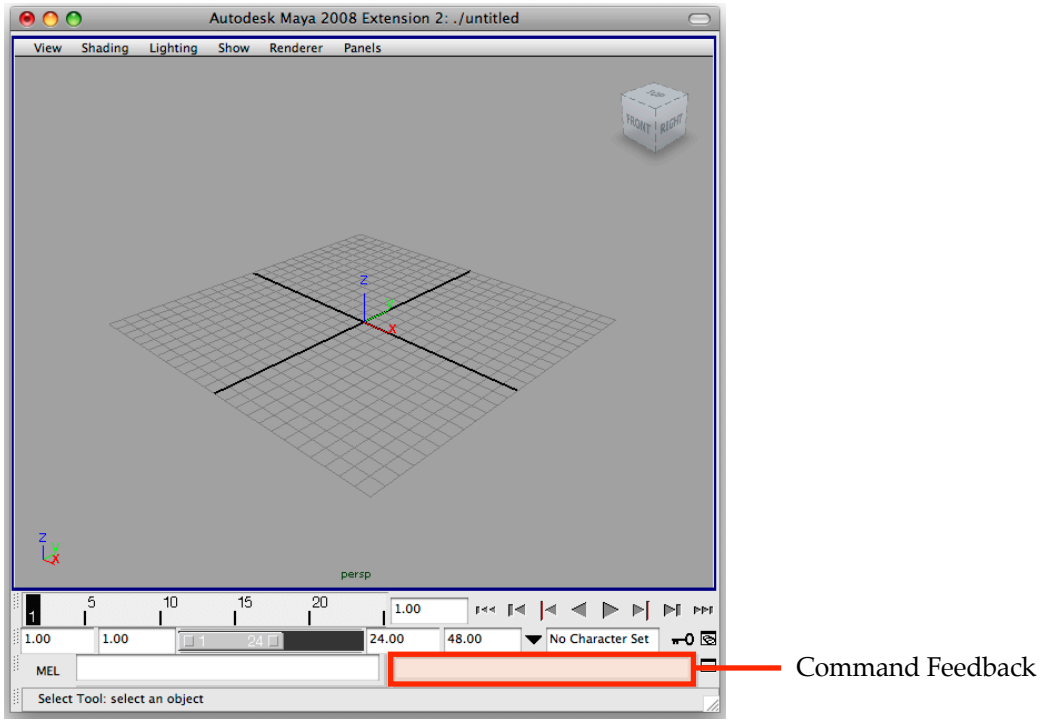


Figura 8.1. El cuadro indica la posición del Command Feedback de Maya.

EL API C++ de Maya tiene funciones que permiten reportar errores y avisos. El método general para reportar mensajes de error es la función `displayError()` de la clase **MGlobal.** Por ejemplo:

```
MGlobal::displayError( “el objeto ha sido borrado” );
```

El mensaje de error es mostrado en rojo en una línea del *Command Feedback* de Maya. Para mostrar un aviso, se utiliza la función `displayWarning()`. Por ejemplo:

```
MGlobal::displayWarning( “el objeto seleccionado es de un tipo
erroneo” );
```

El mensaje de aviso es mostrado en magenta en una línea del *Command Feedback* de Maya. También es posible mostrar información general usando la función `displayInfo()`.

```
MGlobal::displayInfo( “seleccione un polígono” );
```

En Maya, y por tanto para Elesys, no se deben usar ventanas emergentes para mostrar errores y avisos. El mecanismo es utilizar el *Command Feedback*. La razón es que no se puede conocer cuántas veces va a ser llamado un comando, de forma que si se llama repetidas veces, el número de ventanas que puede aparecer pueden llegar a frustrar al usuario.

5. Expresiones Regulares.

Elesys es una aplicación en constante interacción con el usuario. Durante su funcionamiento la introducción de datos y el trabajo con ellos es constante. A pesar de haber normalizado el formato de los datos de entrada, hay que asumir que el error humano siempre estará presente y se trasladará a errores de procesos. Estos pueden venir de la introducción de un dato de formato erróneo, importación de ficheros equivocados, etc.

Los errores causados por la introducción de datos hay que abordarlos y minimizar su incidencia en las tareas de la aplicación. Para el desarrollo de Elesys se ha considerado el mecanismo de **expresiones regulares** como el método más adecuado para tratarlos.

Para tratar las cadenas de datos introducidas por el usuario, mediante teclado o ficheros, se necesita de una rutina de comparación de cadenas. Elesys utiliza cadenas que pueden ser muy complejas además de diversas. Las expresiones regulares es una "elegante" forma de solucionar esto, aplicable en sistemas Unix, sistemas Windows, muchos lenguajes de programación e infinidad de aplicaciones comerciales.

Las expresiones regulares son una serie de caracteres que forman un patrón, normalmente representativo de otro grupo de caracteres mayor, de tal forma que podemos comparar el patrón con otro conjunto de caracteres para ver las coincidencias. Es un sistema cómodo, rápido y potente de realizar un filtrado sobre un determinado caso, y obtener un grupo más reducido y específico, excluyendo los resultados que no coincidan con el patrón dado.

De esta forma, mediante expresiones regulares, podemos componer patrones acordes a los datos que Elesys va necesitando. Esto actuará como filtro a los datos, que serán analizados antes de ser utilizado, de forma que si no se atienen al formato requerido serán rechazados y el programa podrá reportar el aviso sin perder el control en la ejecución.

La figura 8.2. muestra un esquema del proceso de filtrado mediante expresiones regulares que se ha diseñado en el desarrollo del programa.

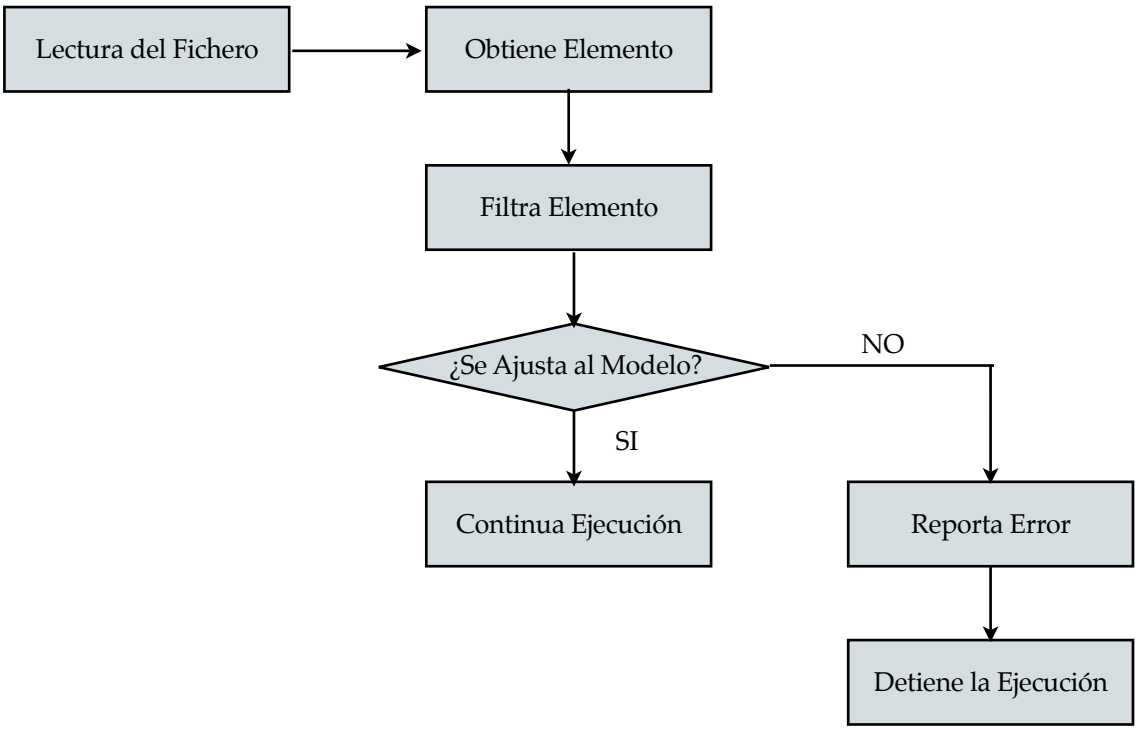


Figura 8.2. Esquema del proceso de filtrado mediante Expresiones Regulares

Teniendo en cuenta que Elesys maneja gran cantidad de parámetros, el uso de estas expresiones regulares es muy usado a lo largo de todo el código. El uso más evidente se encuentra en la lectura de los ficheros de malla y movimiento.

La librería de C++ utilizada es **regex.h**. Ésta se encuentra disponible en las librerías estándar de C++ y es fácilmente utilizable en las tres plataformas para la que se desarrolla Elesys.

Con el fin de describir de forma general los caracteres de control más habituales en expresiones regulares, se adjunta la siguiente tabla explicativa:

Carácter	Descripción	Ejemplo	Interpretación
/	Marca de carácter especial	\é	Busca la letra e con acento
^	Comienzo de una línea	^ae	Empiece cono ae (aeropuerto)

Carácter	Descripción	Ejemplo	Interpretación
$	Final de una línea	to$	Termina con to (aeropuerto)
.	Cualquier carácter (menos el salto de línea)	^.$	Palabras de una sola letra (a, b, c, 2, 7)
|	Indica opciones	pal(a|o)	Busca pala o palo
()	Agrupar caracteres	(vocal)	Busca vocal
[]	Conjunto de caracteres opcionales	[az-AZ]	A hasta Z mayúscula y minúscula
{}	Número específico de caracteres	a{3}	Coincidencia aaa
\w	Cualquier carácter de texto incluyendo _	\w	Cualquier caracter de texto incluyendo _
\d	Cualquier dígito	\d	Desde 0 hasta 9, lo mismo que [09]
-	Separador	[a-z]	Sería lo mismo que [az]
+	La expresión anterior debe estar por lo menos una vez	a+	abc, las, aroma,
*	La expresión anterior puede aparecer cero o más veces	(ab)*	abc, c, ehd

Pongamos como ejemplo la lectura del fichero malla. Hemos aceptado al normalizar el formato, que en la primera línea se debe encontrar un número entero que indique el número de nodos de la figura. Este número debe seguir las siguientes reglas:

- El número debe ser un entero.
- La línea debe terminar con el carácter '|'.

Estas reglas a su vez, profundizando a un nivel más elemental, obliga a que en la primera línea:

- El número debe ser positivo.
- No pueda contener letras.
- No debe tener punto decimal.
- Debe tener al menos un número.
- Debe finalizar con un carácter *pipe* (línea vertical `|`).

Las reglas marcadas en las expresiones regulares se expresan tomando los requerimientos a este nivel más esencial. En este caso, la expresión regular se expresaría como:

```
^[0-9]+[\|]{1}$
```

El código indica que el comienzo de la cadena debe ser seguido de números `[0-9]` , que deben de aparecer al menos una vez `+` . Para terminar debemos encontrar un carácter *pipe* `[\|]` al menos una vez `{1}`. justo antes de la final de la cadena `$`.

Hay que tener en cuenta que ciertos editores de texto, el carácter salto de línea viene acompañado de un retorno de carro. En este caso, la línea no pasaría el patrón, por lo que se debe acompañar de una segunda posibilidad para el caso que el final de la cadena venga con un retorno de carro. Así la expresión regular quedaría:

```
^[0-9]+[\|]{1}$ |^[0-9]+[\|]{1}[\x0D]*$
```

siendo `x0D` el carácter ASCII hexadecimal OD, es decir, el retorno de carro.

Vemos con este ejemplo que, a pesar de ser un requerimiento relativamente sencillo, la expresión de su patrón de expresión regular se complica. Con ello, hay que imaginar los casos en que el patrón deba controlar cadenas de número flotantes (exponenciales o no), cadenas de números y letras alternos, códigos de control, etc.

A cambio de esto, el control sobre los datos importados por Elesys es total. Con ello, es extremadamente difícil que un dato de entrada arruine algún proceso, perdiendo el control de la ejecución. Se puede afirmar que, en caso de existir algún error, éste será provocado por la omisión de algún caso que no se contempló. Con lo que es fácilmente subsanable, modificando el patrón en la etapa de depuración.

Una vez pasado el filtro sólo queda, en caso de no ajustarse al patrón, hacer uso del sistema de reporte de avisos y errores de Maya comentados anteriormente. De esta forma quedaría algo como:

```
// Aplicación de la expresion regular

res = regcomp(&retmp, "^[0-9]+[\x0D]*$|^[0-9]+[\x0D]*[\|]{1}$",
REG_EXTENDED);

res = regexec(&retmp, NumLinPun, (size_t) 1, &mtmp, 0);

// En caso de error reporte a través de Maya

if (res!=0) {

     MGlobal::displayError(MString("El nº no es un nº valido: ") +
NumLinPun);

     return MS::kFailure;

     }
```

Generación De Mallas

El primer objetivo de la aplicación es generar una malla (*mesh*) en función de los datos proporcionados a través del fichero de malla. Estos ficheros contiene la información necesaria para configurar la estructura que se quiere representar.

En el campo de la infografía actual existen varios tipos de malla, cada una de ellas con características distintas. Los tres tipos más utilizados son las poligonales, NURBS y mallas de subdivisión. Es importante conocer las cualidades, ventajas e inconvenientes que aportan cada una para entender la decisión tomada y justificar el por qué esta se adapta mejor a las necesidades del proyecto.

Elesys optará finalmente por mallas poligonales. Éstas se forman a partir de vértices, bordes y caras. Estos elementos constituyentes tienen sus reglas de numeración, sus excepciones y su normas en la composición de los modelos. Todos estos detalles se verán en este capítulo.

1. Mallas NURBS.

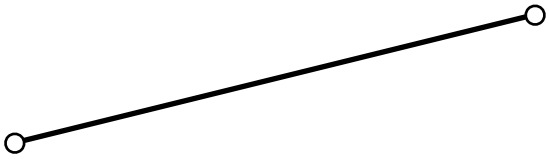
El término NURBS corresponde a las iniciales de *Non Uniform Rational B-Splines* cuyo significado se refiere a un tipo de curvas que forman la base de las superficies del mismo nombre. Estas curvas se denominan *splines* y son, utilizando un simil práctico, el resultado del laminado de una plancha o cinta metálica a través de un tren de rodillos que la van curvando. Estos rodillos constituyen los puntos de conformación metálica o lo que es lo mismo los puntos de control de la curva.

Este símil práctico tiene una representación gráfica matemática mediante una ecuación polinómica del tipo

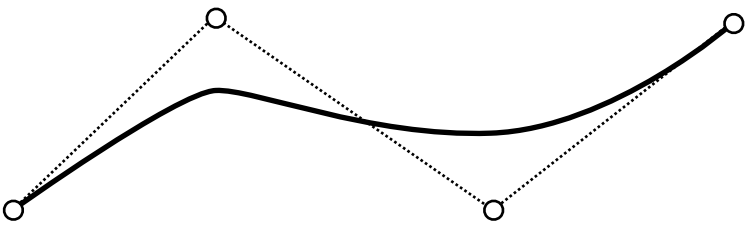
$$y = ax\ n + bx\ (n - 1) + cx\ (n - 2) + \ldots + k$$

donde el exponente constituye el grado de la ecuación o curva. Así la ecuación:

- $y = ax + b$, corresponde a una curva de ecuación lineal de grado 1, que representaría a una recta.



- $y = ax^2 + bx + c$, correspondería a una curva de ecuación cuadrática de grado 2, que representaría a una spline como la de la figura.



Así se podría seguir con las de grado 5 y 7. Por tanto se deduce fácilmente que el grado de la curva corresponde al exponente de la misma y asimismo a su suavidad.

A esta forma de representación se le denomina implícita, ya que cada valor de x le corresponde un valor de y. Pero la representación también puede ser paramétrica, donde se introduce valores de posición de puntos (coordenadas) y para cada punto se determinan los valores x e y. Esta es la forma en la que se basa la herramienta de creación de curvas en Maya. A los valores de las coordenadas de los puntos se les denomina parámetros.

Hay dos métodos de parametrización la uniforme o por longitud de cuerda. El primer método es el que se basa en la introducción de coordenadas de los puntos y es el que utilizan los comandos de creación de curvas CV y EP. El segundo método se basa en que después de la introducción del primer punto, se siguen introduciendo incrementos de longitud de cuerda. Estos incrementos de longitud de cuerda son los que determinan la dirección de la curva..

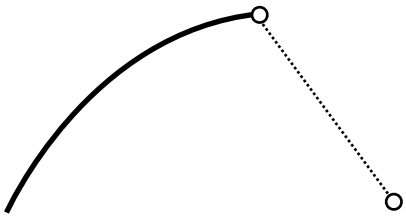


Fig. 7.1. Continuidad de posición (C0)

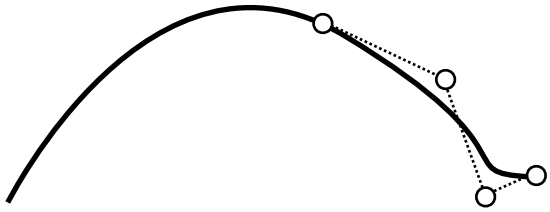


Fig 9.2. Continuidad de tangencia (C1)

Otros aspectos de las curvas NURBS son la *curvatura* y la *continuidad*. La curvatura es la medición del radio de un círculo virtual ajustado al máximo a ella y la continuidad es el tipo de transición entre dos curvas que se juntan, pudiendo ser: de *posición* (C0) y de *tangencia* (C1). Existe una tercera *curvatura* (C3), muy parecida a la de tangencia pero que con su nombre indica con una mayor curvatura.

La característica fundamental del modelado NURBS es que al trabajar con curvas de tipo matemático puede obtenerse de forma directa superficies totalmente suaves, por lo que es muy adecuada su utilización en los modelos orgánicos e inorgánicos que utilicen este tipo de superficies, como podrían ser los cuerpos y caras humanos o los objetos industriales de diseño que utilicen superficies de estilo de textura pulimentada.

En Elesys se trata con cuerpos en los que la posición de cada uno de los nodos en el espacio tiene que ser exacta, no pudiendo estar sometidos a efectos de suavizados por splines, continuidad, etc. que haga que el punto finalmente ocupe una posición diferente a la que marquen los resultados. Esto hace que se descarten las superficies NURBS para la creación de los mallados a analizar.

Si bien esto es cierto, es importante conocer sus posibilidades ya que puede ser interesante utilizarlas en casos de mallado aproximado en los que sea más importante la continuidad y suavizado del modelo que el posicionamiento totalmente exacto de los nodos. Como ejemplo sirva casos en los que se trabaje con sistemas de partículas, fluidos o modelos humanos.

2. Mallas de Subdivisión.

Una superficie de subdivisión es aquella que al aplicarla en un objeto o una selección, subdivide o fragmenta la malla, añadiendo caras, con lo que se consigue que la transición entre ellas se suavice.

La facilidad de trabajo con este tipo de superficies, hace que se pueda partir de un modelo muy simple y de formas bastas, como los bloques de piedra de un escultor y mediante niveles de suavizado, obtener una superficie estéticamente correcta de formas y suavizado, lo que las hace sumamente adecuadas en la creación de personajes y su posterior animación.

La forma de trabajo consiste en crear un modelo inicial base, que se va transformando conforme añade divisiones de superficie y subdivisiones de las mismas, con lo que se consigue unas superficies suavizadas, gracias a la modificación de sus puntos de control en los diferentes niveles de subdivisión, con las ventajas de ello supone y que se resumen a continuación:

- Proporciona un gran control sobre las formas de los modelos.
- Permite crear superficies con doblados o arrugas agudas y de topología arbitraria.
- Grandes posibilidades en animaciones suaves y uniformes gracias a la continuidad de las subdivisiones.
- Posibilidad de aplicación en zonas parciales en modelos complejos.

Considerando este tipo de modelos, estamos ante un caso muy similar a las superficies NURBS. Los suavizados a partir de modelados poligonales pueden alterar la posición exacta de los nodos por lo que no se utilizará en el modelado de los mallados importados en Elesys.

Hay que tener en cuenta que el modelo esta determinado a priori por el fichero de datos, con lo que el usuario no necesita herramientas que le permita componer la malla. La gran ventaja de las superficies de subdivisión es precisamente esta, la posibilidad de crear modelos complejos mediante un modelado inicial sencillo y posterior movimiento de sus curvas de control. Al no tener que realizarse tareas de modelado en Elesys, este tipo de mallado carece de utilidad.

3. Mallas Poligonales.

Una malla poligonal es una colección de vértices, bordes y caras que define la forma de objeto poliédrico en gráficos 3D por ordenador. Son formas o superficies generadas por múltiples vértices (*vertex*) unidos por líneas rectas (*edges*) para formar caras (*faces*). Las caras consisten generalmente en triángulos, cuadriláteros u otros polígonos convexos sencillos, puesto que simplifica la interpretación, pero también puede ser compuesto de polígonos cóncavos más generales o de polígonos con agujeros.

El modelado poligonal nació como una técnica para crear objetos inorgánicos de diseño industrial o arquitectónico por el aspecto regular y rectilíneo de sus formas primitivas, pero los programas han ido mejorando de tal manera que actualmente se pueden generar modelos orgánicos de carácteres o personajes, ya que mediante suavizados y técnicas de subdivisión, pueden conseguirse superficies totalmente parecidas a las pieles humanas.

Las mallas poligonales es un gran campo de estudio dentro de los gráficos por ordenador y el modelado geométrico. Las operaciones que se pueden realizar en este tipo de mallas es muy variada pudiéndose incluir lógica booleana, suavizado, simplificación, etc.

Las mallas volumétricas son mallas distintas a las mallas poligonales puesto que se representa la superficie y el volumen de una estructura, mientras que una malla poligonal solo explicita la representación de la superficie (el volumen esta implícito). Las malla poligonales se utilizan en gráficos por ordenador de forma muy extensa, existiendo algorítmos para mallas poligonales que permiten realizar raytracing, detección de colisiones y dinámica de sólido rígido.

Las mallas proporcionan un tipo de geometría simple y efectiva por el que puede especificar fácilmente objetos de complejidad y topología arbitraria. Esto permite tener un gran control sobre la posición de los vértices dejando la puerta abierta a

futuras operaciones de edición sobre la misma en función de los resultados que queramos obtener en la fase de representación y visualización.

Por ello se considera que es la tipología de malla más adecuada para los propósitos de la aplicación. Su versatilidad, flexibilidad y control sobre sus elementos la sitúan, sin lugar a dudas, como la mejor opción de las estudiadas. Por ello, se realiza a continuación una exposición más detallada de sus elementos y características propias.

3. Vértices

Los vértices de una malla son almacenados como una serie de puntos. La figura 7.3 muestra la cadena de vértices de una malla triangular. Cada punto tienen su propias coordenadas x, y, z.

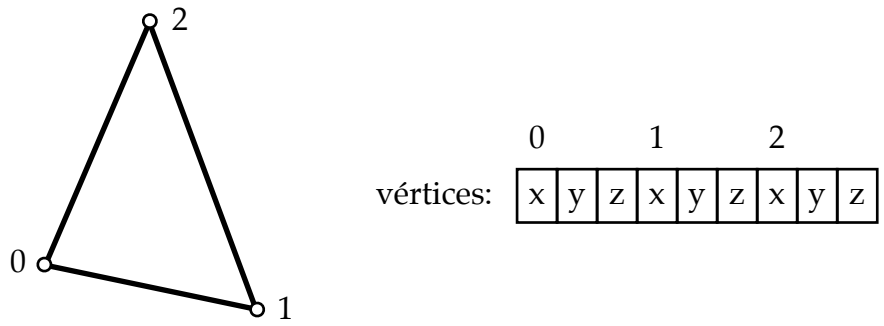


Figura 7.3. Cadena de vértices

Puesto que hay solo tres vértices en un triángulo, la cadena de vértices contiene solo tres puntos. Un elemento en una cadena es accedido mediante su índice. Todos los índices parten del número 0. Para esta malla hay tres vértices, y sus índices son 0, 1 y 2. En general, una cadena tiene índices desde 0 a uno menos del total de los números de elementos.

4. Bordes

Un borde es una línea recta que une dos vértices. Cada línea recta puede ser identificada por su punto inicial y final. Para un borde en una malla, estos puntos son vértices. Así, para identificar unívocamente un borde en particular solamente se necesita dar el vértice inicial y final. Quedando definido por dos vértices. Como se muestra en la sección anterior, los vértices son almacenadas en arrays. Para referirnos a un vértice en particular simplemente necesitamos su índice de la cadena de vértices. Este índice es un número que puede ser almacenado como un entero. Así, para identificar unívocamente un borde necesitamos solo dos enteros: uno para el índice del vértice inicial y uno para el índice del vértice final.

La lista de bordes es un array, en el cual cada elemento consiste en dos enteros. La cadena de vértices y la cadena de bordes para un triángulo se muestra en la figura 7.4.

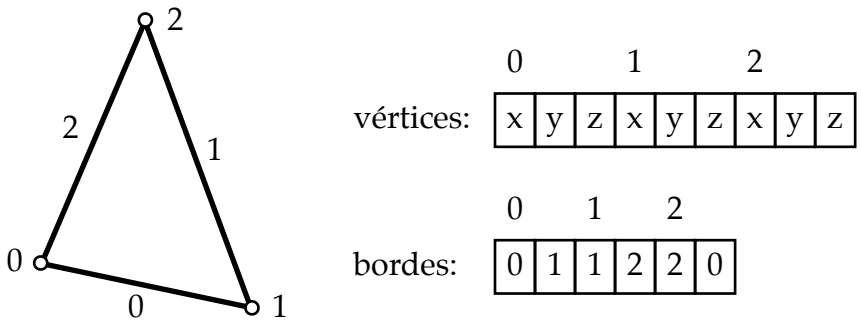


Figura 7.4.. Array de bordes

5. Polígonos

Un polígono es una superficie plana cuyo borde esta definido por una serie de bordes conectados. La figura 7.5. muestra una malla con dos polígonos. Para este ejemplo, ambos polígonos tienen tres bordes y tres vértices, Maya no pone ningún límite al número de bordes o vértices en un polígono. Esto permite crear polígonos con múltiples agujeros.

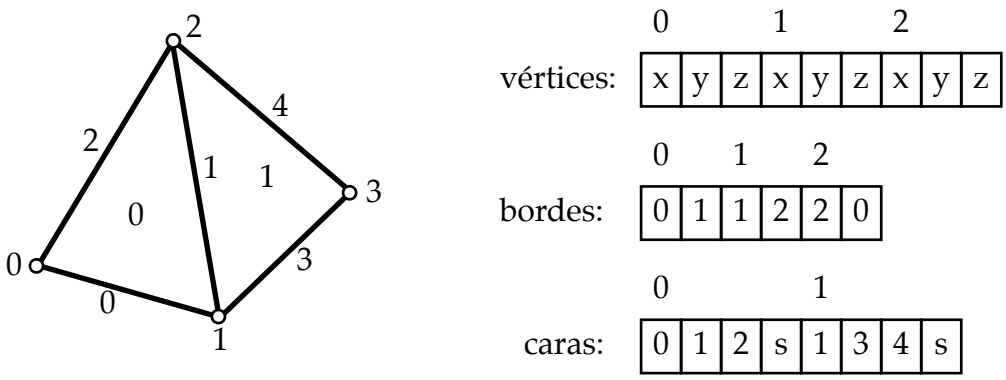


Figura 7.5. Polígonos

Los términos caras y polígonos pueden utilizarse de forma alterna. Maya define una cara en términos de bordes. La cadena de caras contiene una lista con los índices de los bordes que compone la cara. Puesto que un borde puede ser compartido por múltiples caras, puede no ser único apareciendo más de una vez en la lista. La cadena de caras es una continua y larga cadena de índices de bordes. Solo cuando Maya encuentra una cara terminada por el índice *stop*, mostrado como "s", sabe que se ha llegado al final de esa cara y al comienzo de una nueva.

Exponiéndolo de otra forma, el proceso de iteración a través de todas las caras es realizado comenzando por el primer elemento de la cadena de caras e iterando a través todos los elementos hasta que se encuentra el índice especial de bordes. Este índice marca el final de la cara que se esta procesando. El elemento que sigue inmediatamente después es el comienzo del primer borde de la siguiente cara. La iteración continua hasta que se encuentre el siguiente índice de final de cara. Así, este proceso se repite a lo largo de toda la malla. A partir de esta descripción, se intuye fácilmente que encontrar un polígono en particular en una malla con muchas caras tomará mucho tiempo. El iterador se ve obligado a empezar por principio de la cadena cada vez que quiera encontrar una cara en concreto.

Para no verse obligado a emplear este método tan lento para encontrar caras, se genera otra cadena que es utilizada para la búsqueda de caras. Esta cadena es la cadena de *faceIndices* y tendrá tantos elementos como polígonos tenga la malla. Cada elemento de la cadena especifica el índice del comienzo de cada cara. La cadena de *faceIndices* para una malla se muestra en la figura 7.6.

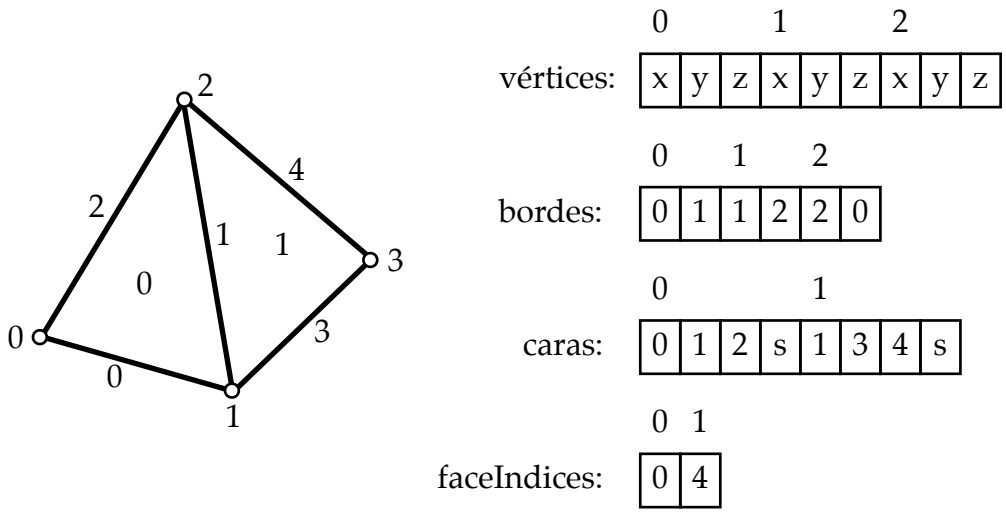


Figura 7.6. Cadena faceIndices

La cadena *faceIndices* tiene un elemento que cuenta con un índice para cada polígono. El primer elemento (índice 0) contiene el índice 0. Esto indica que el comienzo del primer polígono en la cadena de caras está en el índice 0. El segundo elemento (índice 1) cuenta el número 4. Esto indica que el comienzo del segundo polígono esta en el elemento 4 de la cadena de caras. El acceso a las caras así es mucho más rápido.

Para obtener el segundo polígono de la malla, puede iterarse a través de todos los elementos de la cadena hasta llegar a la segunda cara, o bien puede accederse inmediatamente al segundo elementos del *faceIndices* para conocer la posición de la cadena de caras. Al poderse determinar de forma rápida los índices de comienzo y final de cada cara, se puede calcular el orden (número de bordes y vértices) rápidamente.

Afortunadamente, Maya oculta la complejidad de estas cadenas al desarrollador, y en su lugar proporciona funciones apropiadas, más intuitivas, para manipular las caras. Cada polígono es indexado por su posición la lista del número total de caras.

6. Normales

Una normal es un vector perpendicular a una superficie. Con una longitud unidad, es utilizado para determinar la orientación de una cara; es decir, si un lado de una cara se orienta para dentro o para fuera. El uso más frecuente de la normal es la del sombreado de la superficie. Una esfera poligonal, aun teniendo una superficie geométrica con caras muy marcadas, puede ser renderizada como una superficie lisa asignando suavizado normal a la superficie.

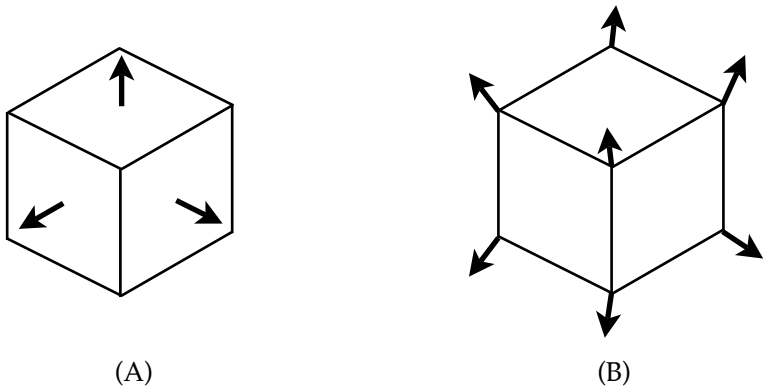


Figura 7.7. Normales a la cara (A), normales al vértices (B)

Maya extiende el concepto de normal permitiendo colocarlas en cualquier dirección. Además, las normales pueden ser asociados a componentes diferentes: a la cara, a los vértices, y a los vértices de cara. La Figura 7.7 (A) muestra las normales de cara para las caras de un cubo poligonal. La normal a una cara también es llamada normal geométrica porque es calculada determinando el vector que es perpendicular a la cara.

También es posible asociar una normal a cada vértice. Estas normales son nombradas como normales a los vértices. Las normales de un vértice para un cubo se muestra en la Figura 7.7. (B). Como la normal esta asociada a un vértice, todas caras que comparten ese vértice pueden usar esa normal.

El último tipo de normales soportadas por mallas poligonales son las normales a los vértices de caras. Esto permite que una normal sea asociada con un vértice dado de una cara. Mediante estas normales asociadas a las caras, se puede tener control de las luces y sombras de cada una de las cara. Las normales a los vértices de caras se muestran en la figura 7.8.

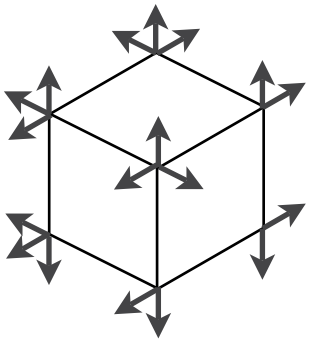


Figura 7.8. Normales a los vértices de caras.

Por defecto, los distintos tipos de normales son calculados automáticamente por Maya. Si los vértices de la malla se mueven, la normales serán calculadas automáticamente.

La normal de cara es calculada para ser perpendicular a la superficie de la cara. Si la cara no es plana, se utiliza una normal calculada como media. Las normales de los vértices son calculadas como la media de las normales de las caras que comparten ese vértices. Las normales de los vértices van hacia fuera desde el vértice, con una dirección que es la media a las normales de todas las caras circundantes.

Es posible que haya las circunstancias en las que se necesite que la normal señale en una dirección concreta. Ambas normales, la de un vértice y la de un vértice de una cara, pueden ser colocadas en una dirección concreta. Esto puede hacerse para simular bordes duros o suaves al renderizar una superficie. Las normales a una cara no puede ser modificadas pero siempre puede modificarse su sentido. Además, la normal a un vértice o a un vértice de cara puede ser bloqueada. De esta forma, Maya no recalculará su dirección nunca más.

7. Mallas Problemáticas

Puesto que una malla se compone principalmente por vértices que son conectados formando caras, es posible crear conectividades que genere mallas problemáticas. Una malla valida no tiene topologías múltiples. Esto significa básicamente que si las caras conectadas de la malla fueran esparcidas fuera del plano no habría pedazos que se superpondrían.

La figura 7.9. muestran tres caras que comparten un borde común. Este escenario es conocido como geometría múltiple conectada. Un borde correcto deberían unir, como máximo, dos caras.

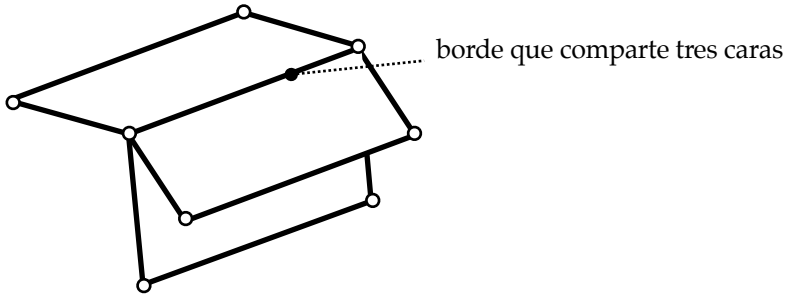


Figura 7.9. Más de dos caras compartiendo un borde.

Otra malla problemática es aquella en la que se comparte un vértice común sin un borde. Esto se muestra en la figura 7.10.

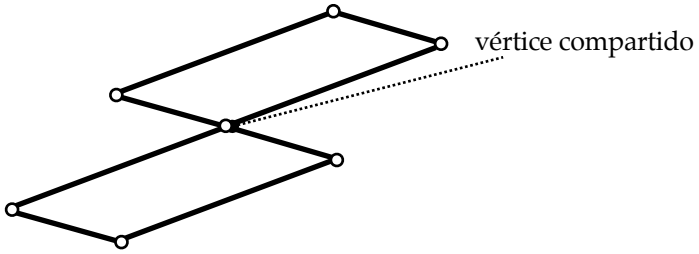


Figura 7.10. Cara que comparte vértice pero no un borde.

Aunque las caras de la malla puedan satisfacer la topología, es posible que las normales no. La Figura 7.11. muestra este caso, donde dos caras adyacentes tienen sus normales en sentido contrario. Esto puede deberse a que se ha especificado los vértices de la cara en orden contrario o que el usuario invierte la normal sin separar las caras.

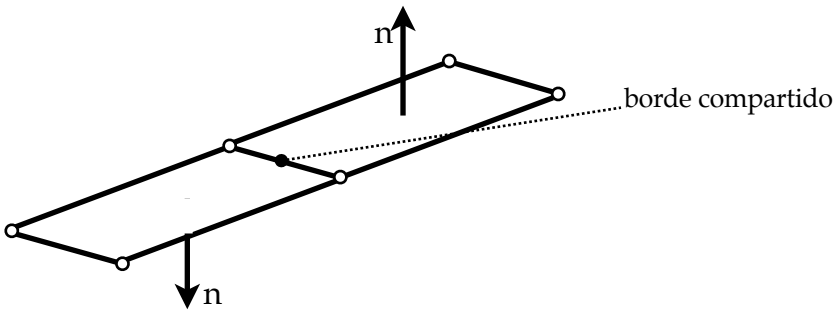


Figura 7.11. Normales invertidas adyacentes.

Aunque estrictamente es válida, debe evitarse las mallas que tengan caras no planas. Una cara con tres vértices (un triángulo) es siempre plana. Si una cara tiene cuatro o más vértices y tiene uno o más de estos que no están en el mismo plano, son considerados caras no planas. Una cara que no es plana, es curva. En este caso no se tendrá una sola normal perpendicular sino que se tendrá la media de las normales en su lugar.

Las caras lámina son también consideradas irregulares. Una cara lámina se pliega sobre sí misma. Esta se puede crear fácilmente duplicando una cara y posteriormente uniendo dos de sus vértices. Aunque son dos caras, estas

comparten idénticos vértices. La superficie se puede entender como "dos caras gruesas". La normal de ambas caras puede señalar la misma dirección o direcciones opuestas.

Aunque las mallas no satisfagan la limitación de dos topología múltiples, seguirán funcionando con muchas de las herramientas de Maya, aunque no con todas.

8. Creación de Mallas

El proceso de creación de una malla consiste en preparar todos los datos necesarios: vértices, caras, etc. Los vértices y las caras son datos fundamentales para la malla, pero opcionalmente se le puede asignar coordenadas de texturas uv, colores, datos blind, etc.

La creación de la malla la realizamos utilizando la clase del API de Maya **MFnMesh**. Esta función tienen un objetivo: asegurar que la malla generada es valida. Maya realiza un chequeo en los vértices y caras que se proporcionan a la función para la creación. Algunos de los chequeos son los siguientes:

- **Vértices repetidos:** si el mismo vértice aparece en la lista de una cara, el duplicado es desechado. Si la lista de vértices de una cara es (1, 2, 3, 3, 4), la lista de vértices de la cara al final sería (1, 2, 3, 4).

- **Vértices no utilizados:** si un vértice no es utilizado en ninguna de las caras, es eliminado.

- **Vértices compartidos:** si una cara usa los mismo vértices más de una vez, se crea un duplicado de ese vértice. Esto asegura que la cara no se colocará detrás de sí misma. Dada la lista de vértices para una cara (1, 2, 3, 1, 4), el cuarto vértice estará duplicado, produciendo un vértice 5. La lista final de vértice en la cara sería (1, 2, 3, 5, 4).

- **Normales no contiguas:** es posible que dos caras adyacentes tenga diferentes orientaciones en sus normales. Si la primera cara tiene sus vértices indicados en orden de las agujas del reloj y la cara vecina tiene también sus vértices indicados en orden de las agujas del reloj, el borde común estará duplicado. Esta duplicación del borde esta hecha haciendo copias de los vértices comunes en el borde y teniendo la segunda cara referenciados a ellas. Esto separa las dos caras ya que tienen sus normales apuntando en distintos sentidos compartiendo un borde común.

Aunque es posible en Maya modelar las mallas con caras láminas y con caras con agujeros, no es posible generar directamente este tipo de mallas usando la función de creación.

9. Clase MFnMesh de la API de Maya.

Elesys utiliza la clase **MFnMesh** para la creación de mallas. Estas mallas tienen unas características muy determinadas y su comportamiento ante el movimiento de sus nodos deben estar definidas de forma especial.

La información contenida en un archivo de malla debe entenderse como la suma de una serie de elementos que se unen para formar un mallado completo. Haciendo un símil, podría compararse con un puzzle. La construcción del puzzle se realiza mediante la unión correcta de sus piezas. En nuestro caso, estas partes constituyentes o piezas son los elementos. La posición, forma y orientación de un elemento viene determinado por sus nodos. Los nodos son las partes más esencial de la malla y el principal parámetro constitutivo de los elementos.

Los elementos a construir pueden tener dos configuraciones: de seis o nueve nodos. Como vimos en capítulos anteriores, al normalizar la constitución del fichero malla, cada elemento tiene un número entero que indicaba el número de

nodos de los que estaba constituido, teniendo que ser siempre este número un seis o un nueve.

A la hora de construir un elemento a partir de la posición en el espacio de sus nodos, el orden con que se tomen esos nodos será determinante. La clase **MFnMesh** recibe los nodos junto con su posición para la construcción de la malla. El orden con el que se entregue los nodos indicará a la función de Maya como debe relacionar posteriormente cada vértices y como debe disponer los bordes que los unan.

La clase del API de Maya **MFnMesh** tiene un comportamiento definido. El método `create` tiene variantes. Entre ellas se ha usado la más adecuada en función de los datos recibidos por el fichero externo de la malla. Esta variante del método `create` acepta una serie de argumentos que comentamos a continuación:

- *Número de vértices:* número entero que especifica el número de nodos que tiene la malla.
- *Número de Polígonos:* Maya denomina polígonos en este caso a las caras formadas por la unión mediante bordes de una serie de vértices. Por tanto, se debe de pasar como argumento el número de relaciones geométricas cerradas que se determinarán en las construcción del modelo.
- *Cadena de puntos de los nodos:* incluye todos los vértices de la malla. Especifica la posición en coordenadas cartesiana de cada uno de los nodos.
- *Cadena de elementos de número de vértices de Polígonos:* especifica el número de vértices con los que cuenta cada polígono del modelo.
- *Indices de vértices de conexión para cada polígono:* cuenta con los números identificadores de los nodos que formarán cada polígono. El número de identificador de los nodos va de 0 a (n-1) siendo n la dimensión de la

cadena de los puntos de los nodos. El número del identificador vendrá dado por la posición que ocupe en la cadena de los puntos de los nodos.

- *Padre o poseedor del polígono:* especifica el padre del modelo creado (si existiera).

- *Valor de estado a devolver:* indica el valor del estado a devolver puesto que el método `create` de la clase **MFnMesh** devuelve un **MStatus**[3]. Este valor servirá para su correcta gestión de errores y reportes de posibles problemas.

10. Configuración de Elementos

Para concretar estas ideas tomamos un fichero de una malla que cuenta con un solo elemento de nueve nodos. El contenido de este fichero sería:

```
9|
1|1.000000|0.000000|0.000000|
2|1.000000|0.500000|0.000000|
3|1.000000|1.000000|0.000000|
4|1.000000|1.000000|0.500000|
5|1.000000|1.000000|1.000000|
6|1.000000|0.500000|1.000000|
7|1.000000|0.000000|1.000000|
8|1.000000|0.000000|0.500000|
9|1.000000|0.500000|0.500000|
1|
1|9|1|2|3|4|5|6|7|8|9|
```

Por su parte, el contenido de un fichero de una malla de un elemento de seis nodos sería:

[3] *Variable de estado de procesos de Maya.*

```
6|
1|0.00000000|0.00000000|0.00000000|
2|-1.0000000|0.50000000|0.00000000|
3|1.00000000|0.50000000|0.00000000|
4|-0.5000000|0.25000000|0.00000000|
5|0.00000000|0.50000000|0.00000000|
6|0.50000000|0.25000000|0.00000000|
1|
1|6|1|2|3|4|5|6|
```

La representación de estos dos mallados puede verse en la figura 7.9. Esta figura muestra la numeración y el resultado final que resulta según la numeración y orden de los ficheros de mallas normalizados.

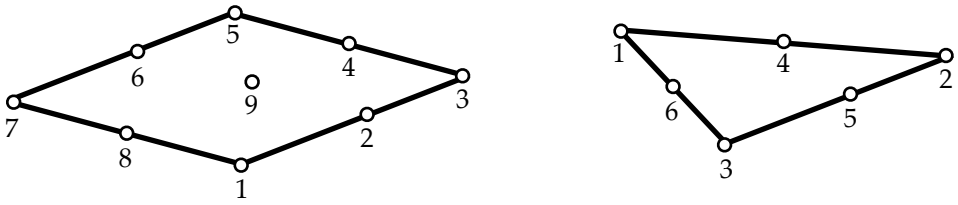


Figura 7.12. Numeración de nodos en elementos

Sin embargo, la numeración y orden que aplica Maya a los elementos creados no coinciden con los aportados por los ficheros. El método `create` de la clase **MFnMesh** necesita tomar los vértices en un orden diferente al que se indica en la figura 7.9.

Una de las tareas principales es el realizar la escritura de los datos en el orden adecuado al **MFnMesh**, partiendo de la información dada por el fichero. Para ello, evidentemente, se ha hecho uso de matrices para el almacenamiento de los datos y punteros adecuadamente programados para el reordenamiento de la información de los nodos.

Para que el proceso sea rápido, se ha hecho un diseño de forma que se optimice el proceso de lectura y escritura de las matrices dinámicas de datos. Se ha creado un algoritmo propio de reordenación. Los resultados obtenidos para grandes mallas, con un gran número de nodos, son del orden de milisegundos. Con ello se evita tiempos de esperas por parte del usuario y la posibilidad de construir mallas de grandes dimensiones sin sobrecargar excesivamente el sistema.

Este reordenamiento debe realizarse de acuerdo a la estructura final que queramos de vértices, bordes, caras y normales en el mallado representado en Maya. Una disposición errónea de relaciones entre nodos (bordes) puede dar lugar a errores en la deformación de la malla. En la figura 7.13. y 7.14. puede verse dos elementos de nueve nodos. El elemento de la izquierda tiene la configuración elegida por Elesys para elementos, la de la derecha muestra una configuración alternativa. Se ve en ambas figuras como, ante un movimiento del nodo central ambos elementos se comportan de forma diferente. Es fácil ver como el comportamiento del elemento de la derecha es anormal, no adaptándose al verdadero movimiento de la malla.

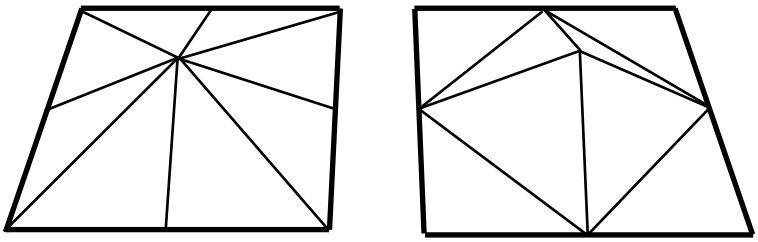


Figura 7.13. Visión frontal de dos elementos de nueve nodos con configuración diferente.

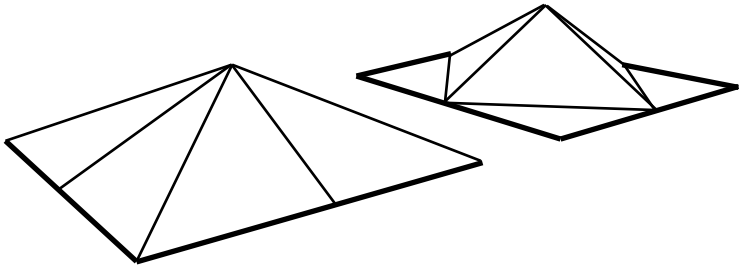


Figura 7.14. Visión lateral de dos elementos de nueve nodos de la figura ASDAS.

Después de un análisis y estudios de los efectos sobre el elemento de variaciones de los nodos en el espacio se ha optado por la configuración de bordes y caras que se puede observar en la figura 7.15.

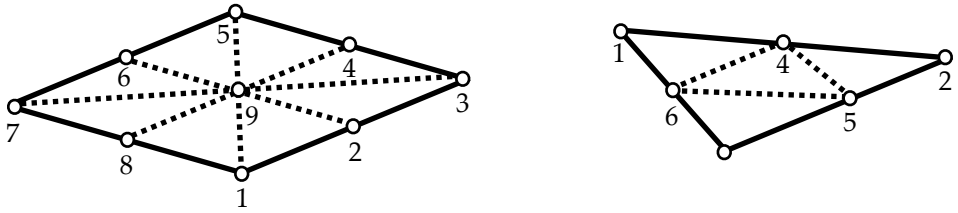


Figura 7.15. Esquema de bordes aplicados a elementos de 6 y 9 nodos.

Se adoptado esta configuración buscando la triangulación de los elementos. La configuración de caras trianguladas da una serie de ventajas que no se tiene en caso de utilizar otra disposición (cuadrada, pentagonal…). La triangulación permite:

- Asegurar la formación de caras planas, sin curvas ni concavidades. Así se evita la posibilidad de formaciones de caras erroneas en mallas de gran complejidad en las que se puedan formar grandes ángulos entre caras.

- Mayor control en las normales a las caras. La posibilidad de contar con caras de tres nodos da gran flexibilidad a la hora de orientar las normales. Con sencillo cambio en el orden de uno de los nodos puede revertir el sentido de la normal.

La elección de mallas poligonales proporciona interpolaciones lineales entre los nodos. La linealidad entre bordes, si bien no asegura la construcción de mallas suaves, si asegurar la exactitud del comportamiento de la malla acorde a las posiciones de los nodos en el espacio. Esta es la principal ventaja de las mallas poligonales a diferencia de NURBS y modelos de subdivisión. Para tener mallas más refinadas se puede acudir a procesos de suavizado.

Figura 7.16. Diferencia entre interpolación lineal y por splines

En la fígura 7.16. se puede observar las diferencias entre la interpolación lineal usada y una interpolación por *splines*, usada en modelos NURBS. Vemos en el caso (b), como la superficie no pasa exactamente por el nodo del centro, sino que queda por la parte inferior. Con ello vemos como no se puede garantizar que la superficie pase por los nodos representados a no ser que se utilice modelos poligonales e interpolación lineal.

La naturalidad de comportamiento de los elementos ante movimientos de sus nodos puede comprobarse en la figura 7.17.

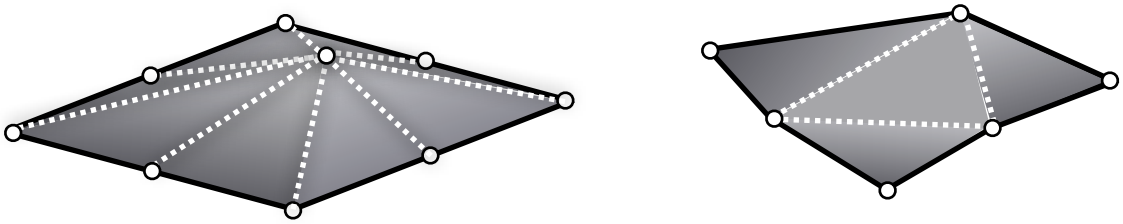


Figura 7.17. Respuesta de los elementos a movimientos de sus nodos.

La deformación de la malla se realiza de forma natural, no obstante debe cuidarse que la representación del problema sea adecuada con un número de nodos no muy bajo. En el caso de no representarse con suficiente detalle, la malla puede presentar picos y una representación poco natural del movimiento.

La figura 7.18. muestra las diferencias de representación entre dos mallados. El primero cuenta un número excesivamente bajo de nodos que no permite su representación fiable. La figura de la derecha, por contra, cuenta con un gran

número de nodos que proporciona una representación más fina del modelo a estudiar.

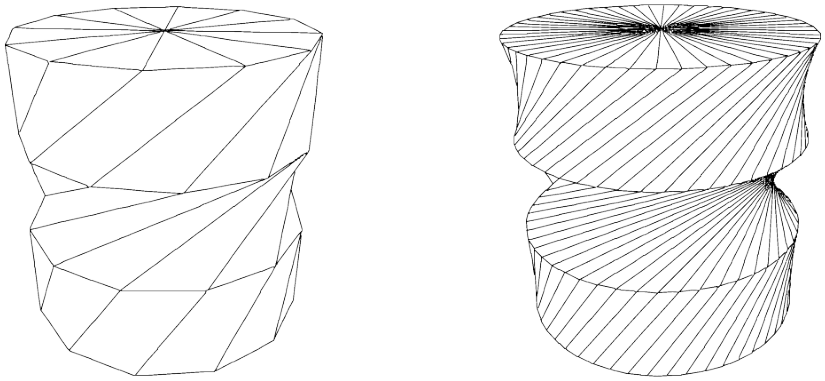


Figura 7.18. Representación de un mismo problema con un mallado con poca precisión (a) y un mallado de gran definición (b).

Generación de Movimientos

En siglos anteriores, los artistas idearon procedimientos para simular escenas en movimiento como dibujar en múltiples hojas una figura con diferentes poses. Al moverlas se producía la sensación de movimiento, si además se movían más deprisa, el movimiento era más veloz, lo cual significaba que el tiempo estaba estrechamente ligado al movimiento, lo que era en definitiva el fundamento de la animación.

De todo ello se dedujo que si se disponía de una serie de imágenes fijas relacionadas entre sí, con pequeñas diferencias jerárquicas y se las movía a una suficiente velocidad, podría llegar a engañar al ojo humano, al simular un movimiento continuo y coherente.

En la actualidad, el cinematógrafo, ejemplo típico de las animaciones, utiliza una velocidad de secuenciación de aproximadamente 24 a 30 cuadros por segundos, según la modalidad elegida entre NTSC, PAL o film. Con esta velocidad se produce un efecto casi real. Cada una de estas imágenes fijas se denominan **cuadros** o **frames**.

En los antiguos dibujos animados que se empezaron a producir para empresas de cinematografía como la mítica Disney, los dibujantes debían realizar manualmente los cuadros uno a uno y con modelos físicos simular todas las posiciones intermedias. Aplicando un simple cálculo, una representación de 1 minuto precisaba unos 2.0000 cuadros. Resulta fácil imaginarse el equipo de dibujantes y el tiempo necesario para completar una película como las producidas en la época.

Gracias a los programas de diseño gráfico y animación, el procedimiento se simplifica ya que el usuario únicamente precisa crear una pose inicial y otra final relacionadas. A estas poses se les denomina **cuadros clave** y a partir de este momento, el **motor de Animación** del programa se encarga de calcular por interpolación los cuadros intermedios que completen la animación.

1. Animación por Keyframe

Es la forma antigua de animar de los artistas dibujantes trasladada al ordenador, consiste en, una vez creado el modelo, por ejemplo un personaje, definir diferentes poses y adjudicarles en el tiempo, sus correspondientes claves o **keys**. Esto quiere decir que para cada clave se han almacenado o memorizado todos los atributos de pose, tiempo y posición. Este almacenamiento se denomina **keyframe**.

El tiempo es un atributo fundamental en una animación y se refiere al tiempo que transcurre entre **frame** y **frame** o lo que en otros términos se podría considerar como una velocidad o rango denominado también **frame rate**. Si la velocidad es la relación entre espacio y tiempo, el **frame rate** es la cantidad de cuadros por segundos que hacen que la sensación de movimiento sea lo más real posible. Los estándares más utilizados en Maya son film (24 f/s), NTSC (30 f/s) y PAL (25 f/s).

Para clarificar la sucesión de términos en principio difíciles de comprender como **keys**, **frames** y **keyframes**, se van a dar unas breves nociones sobre su significado.

- **Key.** Es un término abreviado de **keyframe** por lo que su utilización es indistinta ya que tienen el mismo significado. Cuando se dice que se editan keys se están editando keyframes.
- **Frames.** Corresponde a cada uno del los cuadros representados en una animación.
- **Keyframe.** Es un cuadro al que se le han asignado atributos de tiempo y posición.
- **Keyframe intermedio.** Son los Keyframes creados por el motor de animación del programa mediante el procesado automático de interpolación.
- **Tangent.** Corresponde al vector que indica la pendiente de la curva de animación.

2. Tipos de Animación

La animación tradicional por **keyframes** es utilizada para mover y posicionar mallas en su conjunto. Dentro de la animación por **keyframes**, existe una serie de métodos y sistemas, que buscan en algunos casos, la consecución de deformaciones y movimientos que se ajusten a la física real del modelo, por ejemplo, colisión de cuerpos, deformaciones en músculos, etc. Por otro lado, también se desea una optimización de la cantidad de recursos necesarios para mostrar la representación en tiempo real.

Los sistemas de animación más usuales son:

- **Animación por trayectoria (*path Animation*):** la animación se controla mediante una trayectoria, determinada por una curva de apoyo, que es la que sirve de base para que el objeto describa su movimiento desplazándose por la misma. Un ejemplo se puede ver en la figura 8.1.

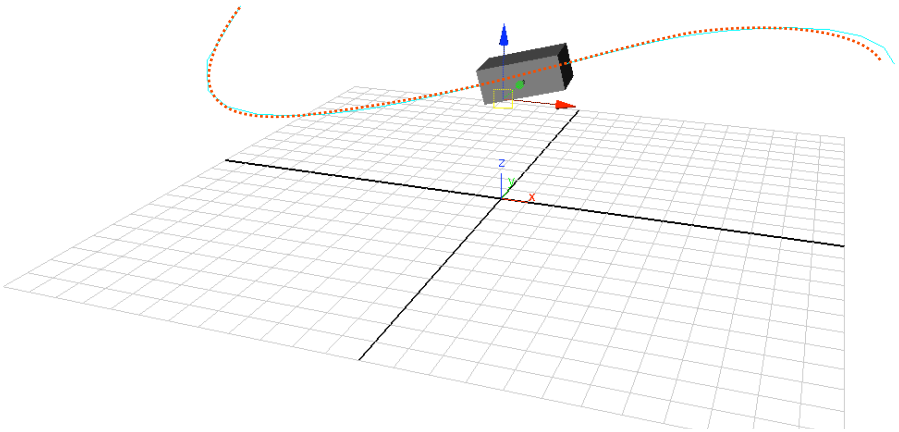


Figura 8.1. Objeto guiado por una trayectoria de animación.

- **Esqueletos (*Skeleton*):** un esqueleto es un grupo de **huesos o *bones***, con nodos de transformación, que se denominan **uniones o *joints*** y que están vinculados entre sí formando una **jerarquía**.

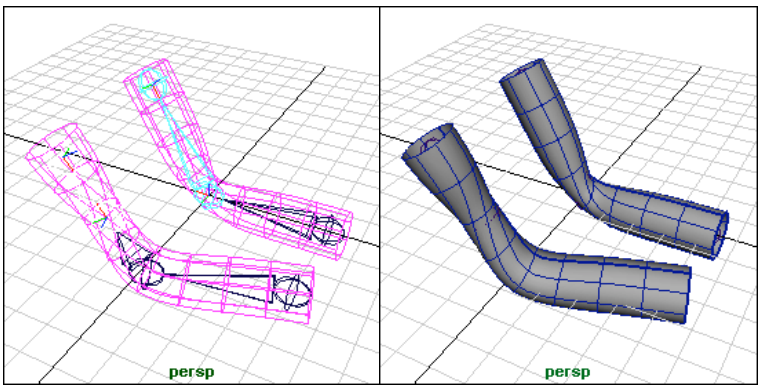


Figura 8.2. Efectos de los Skeleton en dos mallas tubulares.

Se utilizan principalmente para la animación de personajes o criaturas, colocándose en su interior para dirigir sus movimientos, gracias al sistema nervioso o muscular que constituyen sus controladores. Los objetos están relacionados de una forma ascendente-descendente, por lo que sus efectos se transmiten de forma directa. La estructura puede complicarse si se crean **cadenas**, mediante la creación de antecesores y sucesores.

- **Partículas dinámicas (*dynamic particles*):** son sistemas de pequeños objetos que comparten una serie de atributos comunes, que se pueden resumir en: emisión, sincronización y renderizado. Estas características nos permiten definir la situación, la forma, la dinámica y la representación de las partículas creadas. Mediante este sistema puede recrearse elementos naturales como: lluvia, nieve, polvo, humo, fuego, etc.

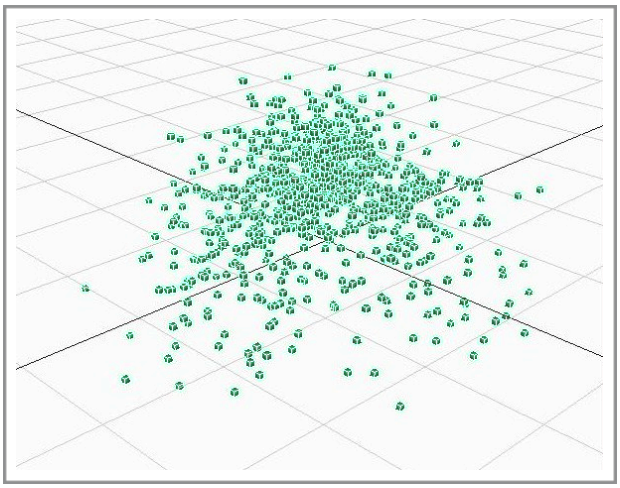


Figura 8.3. Sistemas de partículas en Maya

- **Cuerpos rígidos (*rigid bodies*)** : son aquellos que al ser animados, adoptan una serie de principios físicos que facilitan la simulación real de los movimientos. Estos elementos son sometidos a fuerzas externas denominadas: **fields** o **campos de fuerza**.

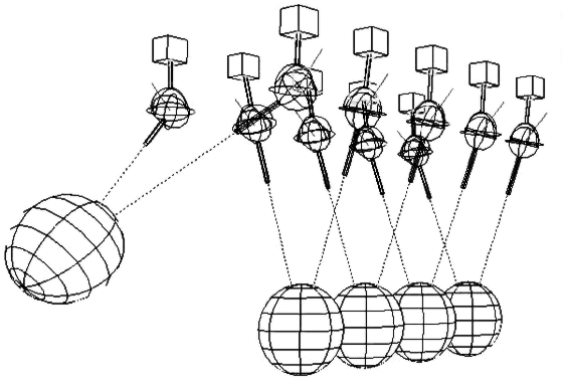


Figura 8.4. Simulación de un péndulo mediante cuerpos rígidos.

Dentro de este sistema los cuerpos se dividen en dos tipos: **activos** o **pasivos**. Los primeros son aquellos que reaccionan directamente mediante la acción de campos de fuerzas y colisiones. En cambio los segundos son aquellos que no reaccionan de ninguna manera a este tipo de acciones.

- **Animación por vértices (*vertex animation*):** consiste en la animación mediante el movimiento directo de los vértices del objeto. Este sistema de animación cuenta con la mayor precisión y control sobre la deformación de la figura pero cuenta con complicaciones extras debido a su necesidad de recursos y potencia de hardware.

3. Obstáculos del Diseño del Sistema de Animación de Elesys

Elesys requiere de una animación en la que se produzcan deformaciones de sus nodos o vértices. Por tanto, de los sistemas analizados en el apartado anterior, parece evidente que tendremos que inclinarnos por la animación por vértices.

Este tipo de sistema de animación es el más flexible de los existentes, ya que permite alta deformación de la malla y gran control sobre ella, al poder manipularse la posición individual de cada vértice en el tiempo. Este tipo de sistema de animación se utiliza normalmente en movimiento científico, que

conlleva deformación del objeto tal y como ocurre, por ejemplo, con la ropa y los fluidos.

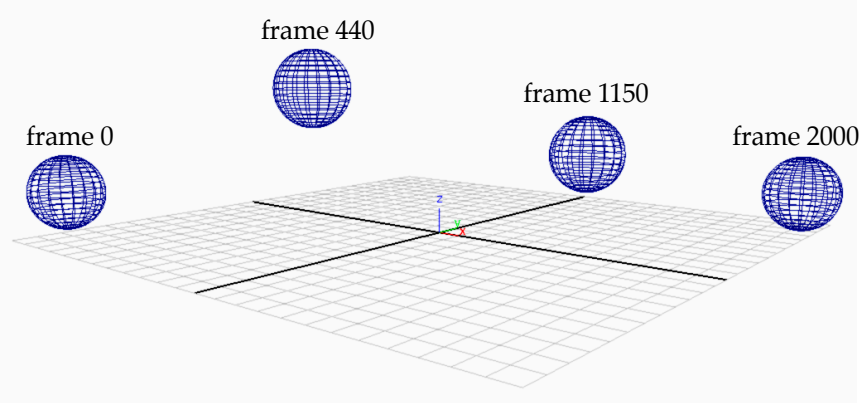


Figura 8.5. Secuencia del movimiento de una esfera de 2000 fotograma con 4 keyframe

Como se comentó, la animación por vértices conlleva una serie de obstáculos relacionados con la complejidad y cantidad de información que se aporta a la computadora, y que esta se ve obligada a procesar a gran velocidad.

Cuando se crea una animación de un cuerpo (pensemos en un mallado sin deformación, sólido rígido) el número de **keyframes** se reduce a uno por fotograma como máximo. Imaginemos, por ejemplo, una esfera que se mueve en una escena. Si la secuencia consta de 2000 fotogramas, tendremos como máximo 2000 **keyframes** en la animación. Incluso este número podría reducirse drásticamente si colocamos **keyframes** claves e interpolamos entre ellos. En la figura 8.5. se observa la animación de la esfera con 2000 fotogramas con solo 4 **keyframes**.

Por otro lado, la *animación por vértices*, consiste en la manipulación directa de los vértices individuales del mallado. Para definir la posición de un vértice en un eje se utiliza 32 bits (coma flotante). Esto hace un total de 12 bytes por vértice (4 bytes por 3 coordenadas).

$$32 \text{ bits} = 4 \text{ bytes} \quad ; \quad 4 \cdot 3 = 12 \text{ bytes}$$

En un animación de vértices, en cada **frame** se guarda la posición de cada nodo. Esto hace que la longitud del fichero crezca notablemente. Si tenemos una pequeña malla de 500 vértices, con 3 coordenadas por vértices y con 2000 **frames** de animación, necesitaremos trabajar en tiempo real con una cantidad de total de 36.000.000 bytes en tiempo real (aproximadamente 36 Mb).

```
12 bytes · 500 vertices · 3 coordenadas · 2000 frames  =  36.000.000 bytes

36.000.000 bytes  =  36 Mb
```

Para el ejemplo nombrado anteriormente, se obliga a Maya a establecer un número de **keyframes**:

```
500 vertices · 2000 frames = 1.000.000 keyframes = 1 millón de keyframes
```

Esto se traduce en Maya en 1 millón de nodos en el DG. Estas cantidades hace que la información generada sea imposible de manejar en tiempo real con las CPU existentes actualmente en el mercado. Además, el uso de la animación tradicional por keyframe es un modo poco práctico y eficiente de trabajar. Queda claro que las necesidades de Elesys necesita de una solución distinta al procedimiento tradicional de animación infográfica.

Aún así, se trató de optimizar el almacenamiento de la posición de los vértices de cada **frame** utilizando métodos como:

- Desechar aquellos movimientos inferiores a un cierto margen definido por el usuario (magnitudes de cm frente a metros por ejemplo).
- Almacenar movimiento lineales o casi lineales como interpolaciones entre el punto inicial y final, desechando puntos intermedios.

A pesar de estos cambios, se continua trabajando con un número enorme de datos que no se puede gestionar. Además, se pierde en exactitud y rigor en los

resultados, más aún cuando hablemos de pequeños desplazamientos o altas frecuencias.

En resumen, utilizando el método tradicional de animación por **keyframe**, se hace imposible manejar con velocidad y eficiencia la animación por vértices que obliga la simulación científica que se tiene por objetivo. Incluso con mallados pequeños, con un número pequeño de vértices, provoca largos tiempo de cargas e incluso bloqueos del sistema por insuficiencia de potencia de la CPU.

Con ello queda patente la necesidad de desarrollar un sistema que permita la animación y visualización de los objetos representados en Elesys en tiempo real, sin periodos de carga, tal y como se estableció en los requisitos escritos en el capítulo 3.

4. Graphics Processing Unit (GPU)

Actualmente el hardware gráfico de las computadoras presentan una serie de características que pueden ser utilizadas. Puesto que se ha comprobado que la técnica tradicional de deformación por keyframe no es válida para el objetivo que nos ocupa, necesitamos técnicas de deformación por hardware que permita la aceleración de la visualización de geometrías de objetos complejos.

GPU es un acrónimo utilizado para abreviar Graphics Processing Unit, que significa "Unidad de Procesado de Gráficos". Una GPU es un procesador dedicado exclusivamente al procesamiento de gráficos, para aligerar la carga de trabajo del procesador central en aplicaciones como los videojuegos y/o aplicaciones 3D interactivas. De esta forma, mientras gran parte de lo relacionado con los gráficos se procesa en la GPU, la CPU puede dedicarse a otro tipo de cálculos.

Diferencias con la CPU

La *GPUs* esta construida con una estructura *highly-parallel* que proporciona un procesamiento más eficiente que unidades de procesamiento central (*CPU*) para un cierto rango de algoritmos complejos. Por ejemplo, uno de estos algoritmos complejos puede ser la representación de un gráfico tridimensional. Una *GPU* puede realiza el número de operaciones gráficas primitivas necesarias, como formar puntos, líneas y triángulos, para crear la imagen compleja tridimensional más rápidamente que lo que lo haría con una unidad de procesamiento central (*CPU*).

Hoy en día, las GPU son muy potentes y pueden incluso superar la frecuencia de reloj de una CPU antigua (más de 500MHz). Pero la potencia de las GPU y su dramático ritmo de desarrollo reciente se deben a dos factores diferentes.

- El primer factor es la alta especialización de las GPU, ya que al estar pensadas para desarrollar una sola tarea, es posible dedicarla para llevar a cabo esas tareas más eficientemente. Por ejemplo, las GPU actuales están optimizadas para cálculo con valores en coma flotante, predominantes en los gráficos 3D.

- Por otro lado, muchas aplicaciones gráficas conllevan un alto grado de paralelismo inherente, al ser sus unidades fundamentales de cálculo (vértices y píxeles) completamente independientes. Por tanto, es una buena estrategia usar la fuerza bruta en las GPU para completar más cálculos en el mismo tiempo. Los modelos actuales de GPU suelen tener una media docena de procesadores de vértices (que ejecutan *vertex shaders*), y hasta dos o tres veces más procesadores de fragmentos o píxeles (que ejecutan *fragment shaders*). De este modo, una frecuencia de reloj de unos 500-600MHz (el estándar hoy en día en las GPU de más potencia), muy baja en comparación con lo ofrecido por las CPU (3.8-4 GHz en los modelos

más potentes (no necesariamente más eficientes), se traduce en una potencia de cálculo mucho mayor gracias a su arquitectura en paralelo.

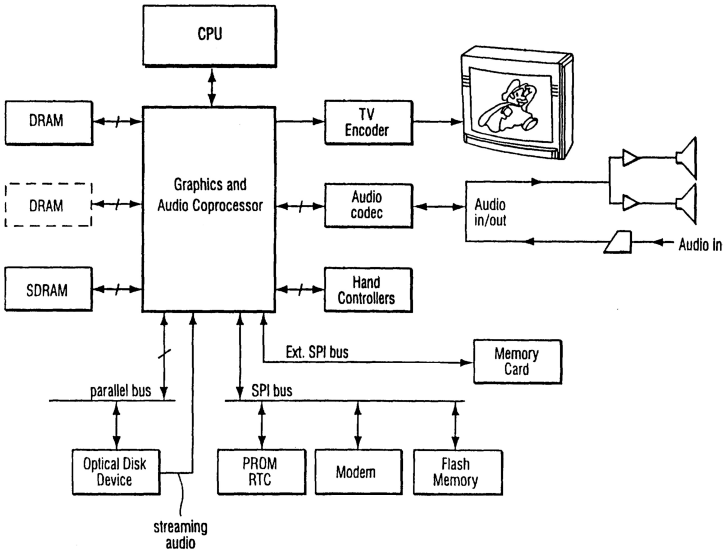


Figura 8.6: Diagrama de bloque de un sistema gráfico interactivo 3D.
Notese la distinción entre CPU y GPU.

Una de las mayores diferencias con la CPU estriba en su arquitectura. A diferencia del procesador central, que tiene una arquitectura *Eckert-Mauchly*, la GPU se basa en el *Modelo Circulante*. Éste modelo facilita el procesamiento en paralelo, y la gran segmentación que posee la GPU para sus tareas.

5. Uso de Memoria Cache

Las ventajas en el uso de la GPU como unidad de procesamiento son obvias conforme a las conclusiones llegadas en el apartado anterior. La velocidad de proceso queda asegurada con el uso de la unidad de procesamiento de la tarjeta gráfica.

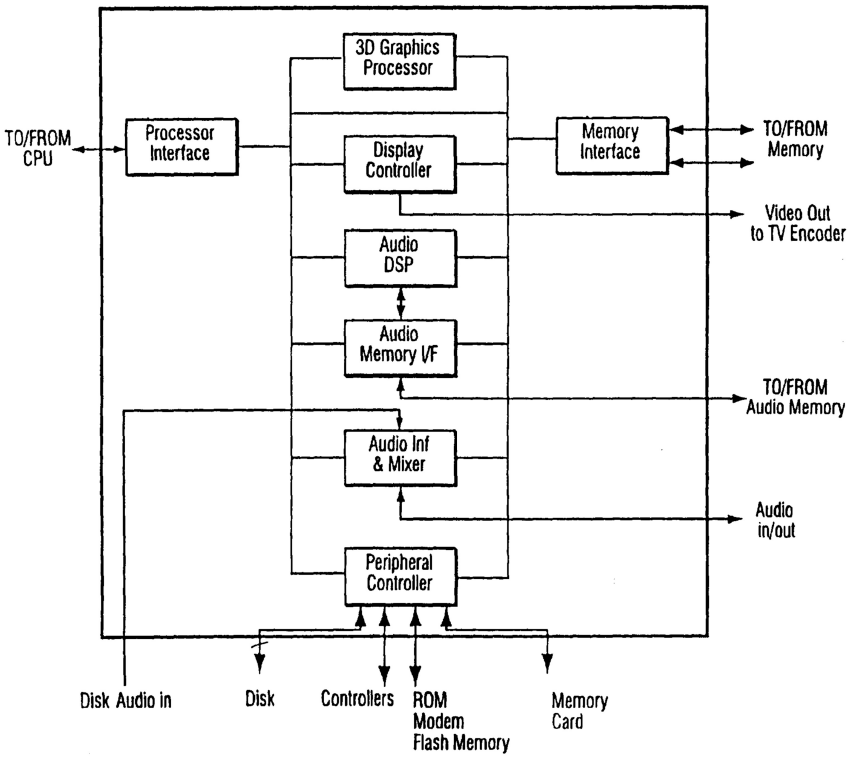


Figura 8.7. Diagrama de un coprocesador gráfico y de audio.

Sin embargo, el verdadero problema a resolver esta en la transmisión de datos. De nada sirve tener gran potencia de cálculo si finalmente, la GPU no dispone convenientemente de los datos con los que ha operar. La velocidad en el intercambio de datos debe ser lo suficientemente rápida como para poder mostrar animaciones de vértices y deformaciones de modelos poligonales en tiempo real.

En los apartado siguientes se analizará las técnicas para procesar imágenes por ordenador a través de una unidad de procesado de gráfico (GPU) utilizando el conocido *vertex cache*. El *vertex cache*, es una memoria caché (sistema especial de almacenamiento de alta velocidad) de la GPU donde se realizan las operaciones relacionadas con la visualización y del posicionamiento de los vértices de un modelo geométrico.

En Elesys, los datos de los vértices poligonales son almacenados mediante el *vertex cache*. Los polígonos son representados como cadenas indexadas. De esta forma se indexa una lista de datos de alguna característica de los vértice como, por ejemplo, las posiciones, los colores, las normales a la superficies o las coordenadas de las texturas.

Elesys usa el *vertex cache* para que la GPU tenga en todo momento indexados y disponible los datos básicos de los atributos de los vértices. De esta forma se proporciona la localización espacial de cada punto, de forma que se puede procesar la escena sin que se requiera almacenar previamente los datos en el orden que requiera la presentación.

6. Diseño de Datos para el Vertex Cache

En Elesys es importante poder representar eficientemente los modelos poligonales existentes en una determinada escena. Para ello es deseable que el programa ponga a disposición de la GPU, lo suficientemente rápido, la información necesaria para una tarea particular.

Podemos caracterizar los datos en términos de **localidad temporal** o **localidad espacial**. Unos datos tienen alta localidad temporal si son llamados frecuentemente por el sistema en un pequeño periodo de tiempo. En general, la representación poligonal de los datos por una aplicación gráfica interactiva 3D tiene un gran grado de localidad temporal.

Por otro lado, la localidad espacial significa que el siguiente dato a referenciar se almacenará en memoria cerca del último dato que se referenció anteriormente. Se pueden conseguir mejorar la eficiencia aumentando la localidad espacial de los datos. En la práctica el rendimiento se puede aumentar considerablemente si, todo los datos que se necesitan para realizar una tarea dada, son almacenados cerca unos de otros en una memoria de baja latencia.

Para aumentar la localidad espacial, se pueden ordenar los datos de los polígonos en base al orden de procesado. Así se asegura que, todo los datos necesarios para realizar una tarea concreta, serán enviados de forma que podrán ser almacenados juntos. En el caso de Elesys, los datos de los polígonos necesarios para animar el modelo pueden ser ordenados según el tipo de animación que se vaya a realizar. Cuando por ejemplo, estamos ante una animación en tiempo real típica de deformación de una superficie, se requiere la manipulación de todos los vértices de la superficie. Para realizar esta animación eficientemente, es deseable que se ordene los datos de los vértices de una cierta forma.

Normalmente los sistemas de gráficos 3D realizan el proceso de animación y representación de forma separada, y en estos pasos separados procesan la información de forma diferente. Desafortunadamente, el orden óptimo de los datos de los vértices para su procesado en tareas de animación es generalmente diferente del orden óptimo para un proceso de representación del modelo poligonal. Ordenando los datos para la animación se puede tender a agregar aleatoriedad al orden de representación. Ordenando el flujo de datos para simplificar el proceso de animación, hacemos más difícil el proceso para su representación.

Así, por varias razones, puede que no sea posible asumir que exista localidad espacial cuando se acceda a los datos para su representación. La dificultad surge cuando hay necesidad de conseguir acceso de forma eficiente a un elemento concreto de un modelo enorme. Además, por razones explicadas anteriormente, habrá normalmente una cantidad de aleatoriedad en la disposición de los datos, al menos para propósitos de representación, con respecto al orden óptimo que deberían presentar para su uso en el motor gráfico. También hay que considerar que es posible que haya otra localidad de datos por encima del nivel de vértice que podría ser útil implementar (por ejemplo, agrupando todos los polígonos que comparten cierta textura).

Un enfoque para mejorar el proceso es la de proporcionar mayor memoria adicional de baja latencia (por ejemplo, a través de adquirir un sistema de memoria de menor latencia). Sin embargo, no se puede asegurar que el tamaño del modelo no rebase estos límites y necesiten un doble almacenado. Esto obligaría a un continuo tránsito entre los datos almacenados en la memoria de baja latencia, y los datos sobrantes que no hayan podido entrar por falta de espacio y se necesiten para realizar el proceso. Por lo tanto, los buffers necesarios para este enfoque podrían ser muy grandes.

También puede plantearse el uso de los datos de caché de la CPU para tomar, reunir y almacenar los datos de los polígonos en un orden óptimo para el motor de representación. Sin embargo, para hacer esto de forma eficaz, se tendría que proteger de alguna forma los datos del poligono de un conflicto del resto de los datos del cache. Además, al tenerse que tomar los datos a almacenar en la memoria latente previamente, pudiera haber algún tipo de aleatoriedad en la manera que son ordenados en la cache de la CPU. Es evidente que el orden, difícilmente sería el óptimo para un posterior uso por parte de la GPU. Este enfoque podría suponer una sobrecarga de la CPU. Usando este método, la CPU central y el motor gráfico tendría que estar conectados en serie, con la CPU alimentando de datos directamente al motor gráfico. Si paralelizaramos el proceso (por ejemplo, alimentando al motor gráfico a través de un buffer DRAM FIFO) se requeriría un aumento sustancial del ancho de banda del acceso a memoria comparándolo con el modo inmediato de alimentación.

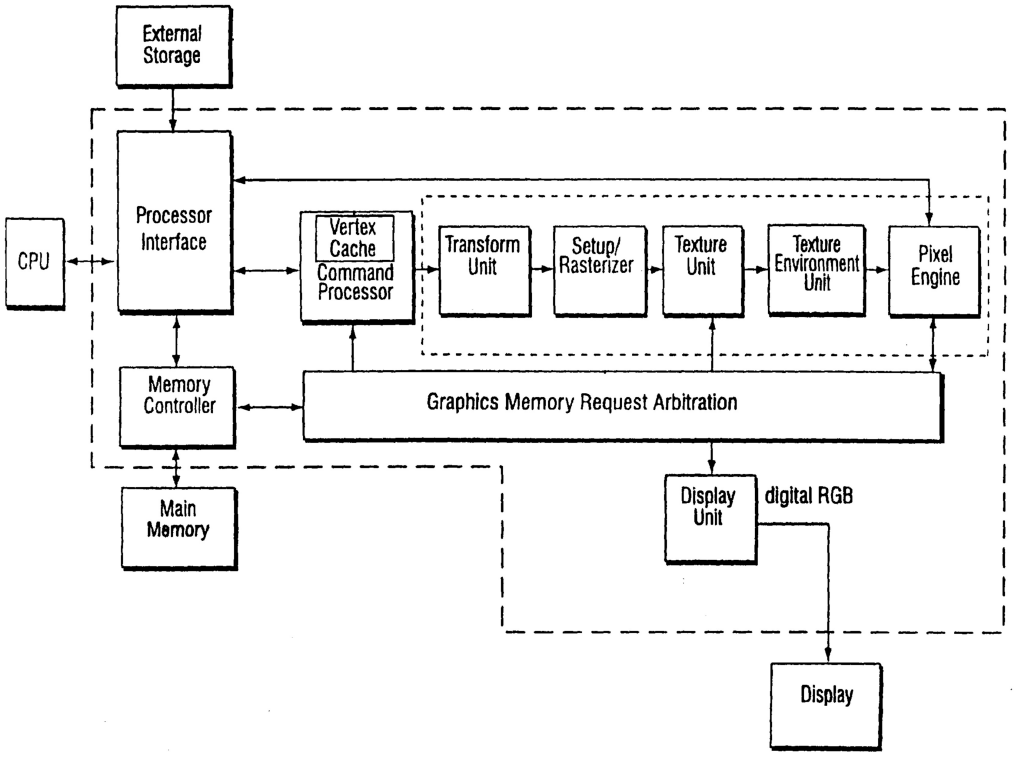


Figura 8.8. Diagrama esquemático más detallado de porciones de la figura 1A del coprocesador gráfico y de audio donde se muestra un ejemplo de procesado pipeline 3D

Así, existe una necesidad de utilizar una técnica más eficiente que pueda ser usada para representar, almacenar y entregar datos de polígonos para realizar procesos de representación y animación de gráficos 3D. La solución adoptada soluciona este problema proporcionando un **cache de vértices** (*vertex cache)* para organizando, mediante indexación, flujos de datos de vértices.

De acuerdo con la solución adoptada, el motor gráfico se alimenta de datos de vértices del polígono por medio del *vertex cache*. Por medio de este cache se obtiene gran flexibilidad y eficiencia que proporciona una memoria virtual mucho mayor a la del actual contenido del cache.

El *vertex cache* puede ser usado para construir el conjunto de datos de vértices necesarios para la representación y animación del modelo en tiempo real. Así, en lugar de ordenar previamente los datos de los vértices para fines de representación gráfica, el *vertex cache* simplemente toma los bloques relevantes de datos para hacer que estos estén disponibles para el procesador gráfico GPU.

El *vertex cache* indexa los vértices del polígono como se ve en el ejemplo de la figura 8.9. En este ejemplo, se indexa distintos aspectos de los vértices del polígono como su posición, color, textura, etc. El *vertex cache* accede a los datos de los vértices que necesita por medio de estos arrays localizados en la memoria principal (*main memory*). De esta manera tiene un acceso rápido a los datos para posteriormente transferirlo al *Display List Proc.*

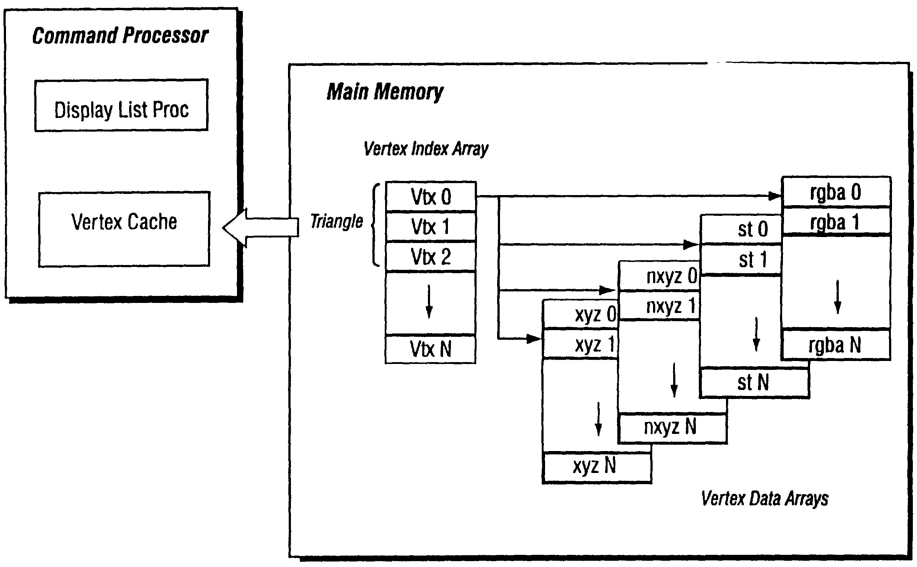


Figura 8.9. Esquema del procesador de comandos incluido vertex cache con cadena de vértices indexados.

Hemos encontrado que, basado en el alto grado de la localidad temporal exhibida por los datos de vértices en Elesys y el uso óptimo de determinadas cadenas indexadas de estructura de datos, la mayor parte de los datos de vértices necesitados en algún momento estarán disponibles en un pequeño conjunto

asociativo del *vertex cache*, teniendo un número de líneas de cache proporcionales al número de flujo de datos de vértices.

La eficiencia puede aumentarse si se crean unas etiquetas configuradas adecuadamente que se asocien a los vértices en el *vertex cache*. La información de estas etiquetas se usará para entregar los datos de los vértices al motor gráfico. Este diseño a medida de las etiquetas permitirá tomar y ensamblar los vértices más rápidamente que con una configuración general.

Puesto que el *vertex cache* alimenta directamente al motor gráfico, se evita el coste adicional de memoria. El acceso directo a la memoria puede ser usado para transferir eficientemente los datos de los vértices dentro de *vertex cache*.

Para aumentar aún más la eficiencia proporcionada por el *vertex cache*, es deseable reducir completamente la necesidad re-etiquetar un polígono o un conjunto de polígonos o el conjunto particulares de polígonos cada vez que son utilizados. De acuerdo con esto, los polígonos pueden ser representados como arrays, como listas de datos que representan alguna característica de un vértice (por ejemplo, las posiciones, los colores, de superficie normal, o las coordenadas de la textura). Cada objeto mostrado puede ser representado como una colección de estos arrays junto con conjuntos de índices. Los índices referencian los arrays para un propósito particular de la animación o representación.

Mediante la representación del los datos del polígono como una lista de componentes indexados, es posible que los datos sean almacenados de forma más compacta (por ejemplo, en un formato comprimido).

Mediante el uso de vértices indexados para la representación junto con el con el *vertex cache*, no hay necesidad de proporcionar ningún otro recurso para fines de visualización. Por ejemplo, se pueden enviar al motor gráfico datos de vértices convenientemente ordenados para realizar la tarea de animación. Esto hace que la animación se procese más eficiente. Por su parte, para representar la malla,

basta con que el *vertex cache* utilice los datos de vértices indexados para la representación, datos que serán ordenados de forma distinta a la animación, para proporcionar de forma más eficiente los datos de vértices necesarios para el motor gráfico para realizar esta tarea.

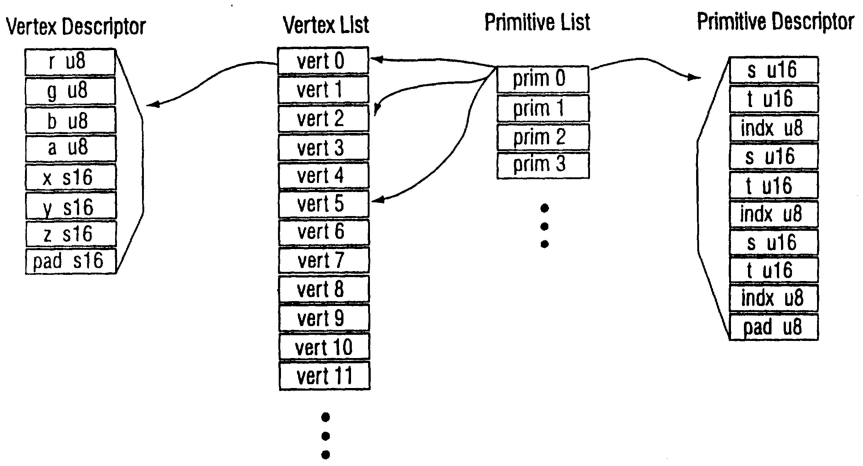


Figura 8.10. Diagrama de una estructura de vértices indexados

La figura 8.10. muestra más detalladamente un ejemplo de una lista de vértices indexados. Esta lista se usa para proporcionar un acceso indirecto a los datos a través del *vertex cache*.

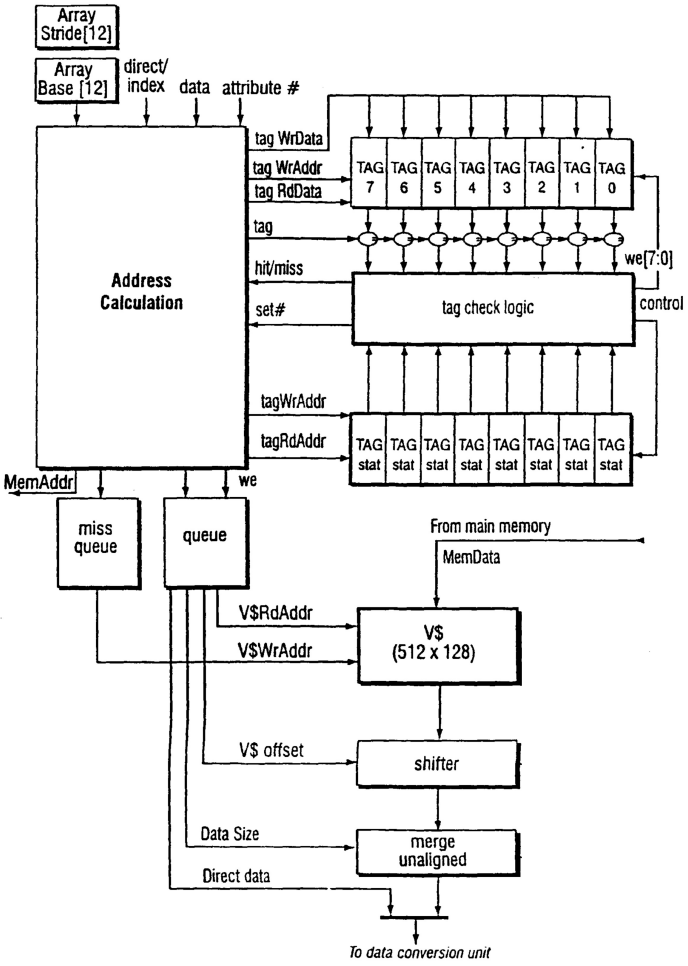


Figura 8.11. Diagrama de implementación del vertex cache

En la figura 8.11. se muestra un ejemplo de diagrama esquemático de un *vertex cache* y su logica asociada. El *vertex cache* de este ejemplo incluye una memoria de 8 kb de caché organizada como 512 x 112 bit dual RAM.

Cualquier aspecto del vértice puede ser indexado o introducido directamente en el flujo de comandos. Esto permite un procesamiento de datos más eficiente por la CPU sin necesitar que su salida cree la estructura de datos del necesaria por el motor gráfico.

Por otro lado, no hay penalización por ordenar los datos de los vértices en el orden de representación; los datos de los vértices son eficientemente presentados por el motor gráfico en cualquier caso, sin que el *vertex cache* degrade

significativamente el rendimiento de la presentación de la estructura de vértices optimizada para presentar los datos ordenados previamente para ser mostrados.

7. Caché de Geometría

Elesys realiza la animación utilizando la tecnología *vertex cache* descrita en el apartados anterior. Hemos visto, que en una lista de vértices indexados puede referenciarse múltiples atributos del vértice.

De todos ellos, la animación exige que la GPU tenga acceso a la posición x, y, z de cada uno de los vértices del polígono para cada instante de tiempo. Por tanto será este atributo el que tendrá que ser convenientemente indexado y proporcionado al *vertex cache* para su proceso, creando un *cache de geometría* (*geometry cache*).

Discretización del tiempo

Como expusimos en apartados anteriores, el movimiento de los puntos importados vienen dados en el dominio de la frecuencia. El dominio de la frecuencia permite obtener la posición x, y, z de cada punto en cualquier instante de tiempo. Resulta un formato válido para representar este tipo de movimientos periódicos. Sin embargo no resulta la forma más adecuada de escribir los datos para su representación en Maya.

Puesto que estamos ante un movimiento periódico, se podría optar por calcular la animación concerniente a un ciclo, para posteriormente mostrarse en bucle. Sin embargo surge el problema provocado de la discretización del tiempo en Maya. Como se comentó anteriormente, Maya muestra instantáneas del modelo de forma que, al reproducirse a una velocidad lo suficientemente alta, el cerebro humano percibe como un movimiento continuo.

De esta forma solo se calculará la posición de cada punto para una serie de instantes o fotogramas. Estos serán posteriormente reproducidos a una velocidad determinada, normalmente 24 o 25 fotogramas/segundo. Resulta complicado y costoso tratar de ajustar la duración real de un ciclo a la escala discreta de fotogramas. Es improbable que un ciclo se complete en el instante exacto de representación de un fotograma.

Como se ve en la figura 5, si discretizamos un movimiento cíclico en intervalos de 24 fotogramas por segundo, tendremos los instantes indicados con línea punteada azules. Vemos que es estadísticamente muy improbable que una de esas marcas caiga exáctamente en el punto marcado como principio y final del bucle PF. Por tanto, el bucle no se cerraría exactamente, produciéndose saltos y comportamientos extraños cuando la animación sea reproducida.

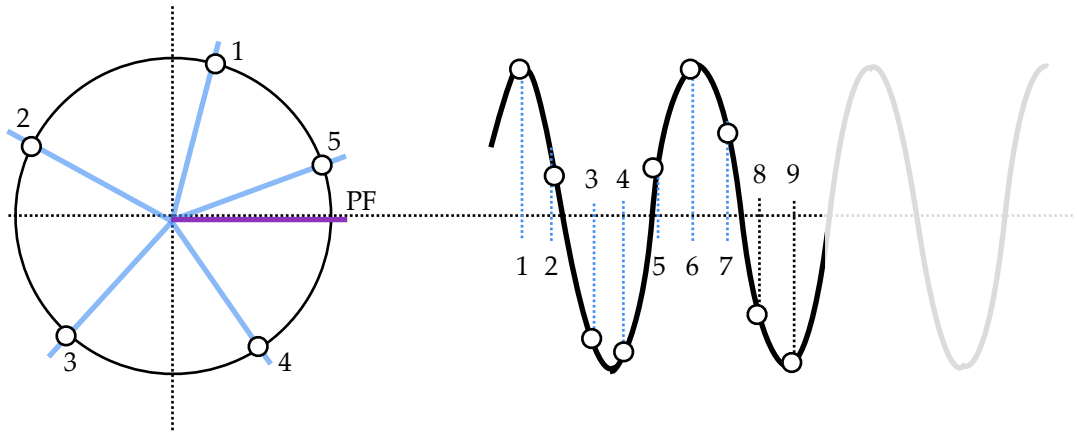


Figura 8.12. Discretización de un movimiento cíclico de frecuencia determinada en intervalos de 24 fotogramas.

Si a esto unimos que el costo de tiempo para calcular cada fotograma es casi despreciable, es evidente que resulta mucho más aconsejable representar todo el rango de tiempo que el usuario necesite visualizar. La interfaz de usuario se ocupará de limitar el tiempo de animación a cantidades razonables, de forma que no se desborde el proceso de cálculo en el caso de solicitar un tiempo de animación excesivo.

Del Dominio de la Frecuencia al Dominio del Tiempo

Maya trabaja principalmente en coordenadas cartesianas y tiempo en fotogramas, aunque puede trabajar en otros sistemas de referencia y en unidades de tiempo. Por tanto, en primer lugar, se tendrá que hacer una conversión del dominio de la frecuencia al dominio del tiempo de los datos proporcionados.

Los ficheros de movimientos cuentan con la parte real e imaginaria del movimiento en cada uno de los ejes coordenados. Si particularizamos para el eje X, la variación de cada punto desde su posición estática vendrá dada al solucionar la siguiente ecuación:

$$\Delta X = Mod \cdot \cos(\omega t)$$

siendo

- **Mod.:** el módulo en el eje X del movimiento que vendrá dado por:

$$Mod = \sqrt{Re_X^2 + Im_X^2}$$

- **ω:** la frecuencia del movimiento estudiado.
- **t:** tiempo en segundos.

Para obtener la posición en el eje Y, Z se procedería de igual forma, tomando la parte real y parte imaginaría en el eje correspondiente.

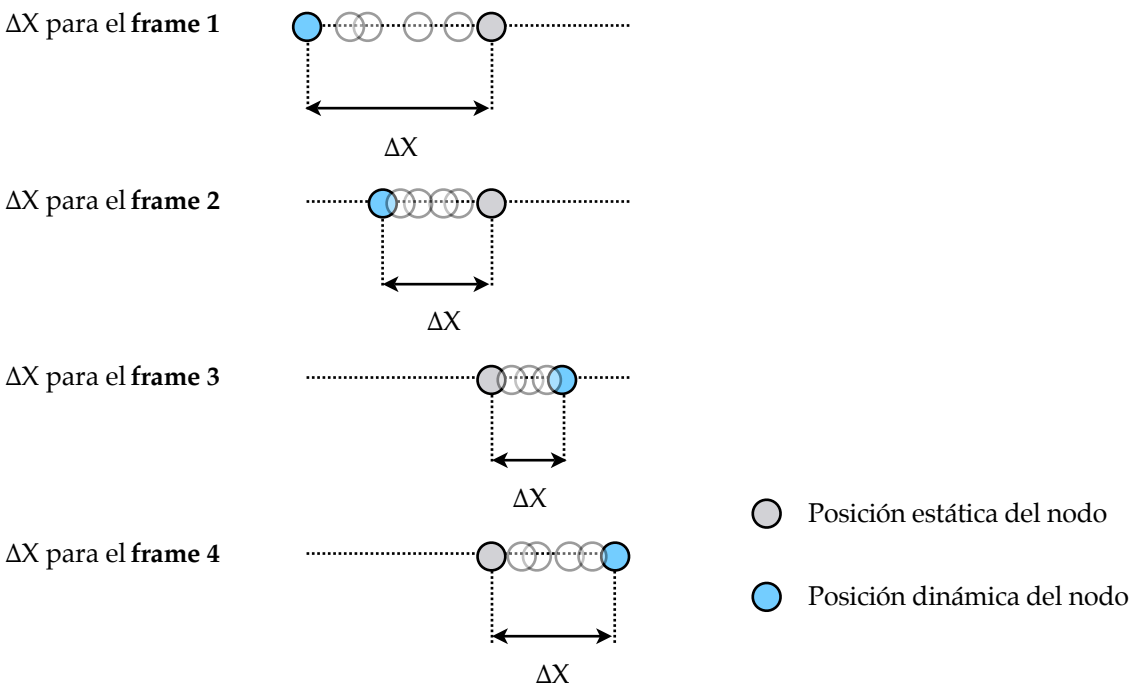


Figura 8.13. Esquema de la variación ΔX del nodo en torno a su posición estática de equilibrio.

Como vimos en el apartado anterior, Maya trabaja discretizando el tiempo en fotogramas. Por ello debemos realizar la conversión de unidades entre segundos (tiempo continuo) y fotogramas (tiempo discreto). No hay que olvidar que el objetivo final de los cálculos no es obtener la posición para cualquier instante de tiempo. Se trata de conseguir la posición de cada vértice del polígono en una serie de instantes muy determinados.

Es muy importante seleccionar el número de fotogramas por segundos que se necesitará en la animación resultante. Desde el punto de vista de la visualización en una computadora, este dato no es muy importante siempre que se encuentre entre unos valores acotados, más exactamente, entre 15 y 30 fotogramas por segundos. Por debajo de 15 fotogramas, la visualización se muestra con poca sensación de continuidad, con movimientos poco suaves. Por encima de 30 fotogramas, el proceso sobrecargaría de forma innecesaría al sistema puesto que no aportaría ningún tipo de ventajas y obligaría a calcular y representar un número adicional de fotogramas por segundos.

Desde el punto de vista de la creación de un video para su visualización de alta calidad, el tema es más delicado si cabe. Es importante conocer la finalidad y el soporte que se utilizará para su representación. En el caso de destinarse para su reproducción en medios audiovisuales como televisión, DVD, etc. Se deberá atender a los estándares PAL y NTSC:

- **PAL** es la sigla de *Phase Alternating Line* (en español línea alternada en fase). Es el nombre con el que se designa al sistema de codificación empleado en la transmisión de señales de televisión analógica en color en la mayor parte del mundo. Es de origen alemán y se utiliza en la mayoría de los países africanos, asiáticos y europeos, además de Australia y algunos países latinoamericanos.

  En el caso de reproducirse en dispositivos PAL, todo depende del país a reproducir. Por ejemplo, mientras en Europa el número de cuadros por segundos es de 25, en países sudamericanos como Brasil este número aumenta hasta 29,97. Todo depende de la modalidad del tipo PAL utilizada (B/G, D/K, -I, -M , -NC, -N).

- **NTSC** (*National Television System Committee*, en español Comisión Nacional de Sistemas de Televisión) es un sistema de codificación y transmisión de Televisión a color analógica desarrollado en Estados Unidos en torno a 1940, y que se emplea en la actualidad en la mayor parte de América y Japón, entre otros países. Un derivado de NTSC es el sistema PAL que se emplea en Europa y países de Sudamérica. La variación de cuadros por segundos es menor que en el sistema PAL. Normalmente, puede utilizarse 29 - 30 fotogramas por segundos (29,970 f/s).

Esto explica la importancia de determinar correctamente la tasa de fotogramas por segundos. Una vez elegido el parámetro podemos desglosar la variable **t** de tiempo en segundos para expresarla en fotogramas por segundo como:

$$t = T_T \cdot N_F$$

siendo: T_T = tasa de cuadros por segundos

N_F = Número del frame a representar

Así, Elesys toma los datos y comienza por discretizar la posición x, y, z de cada punto, empezando por el primer fotograma. Con el fin de clarificar ideas, concretaremos con un ejemplo.

Si tomamos una línea del fichero de movimiento, referente al punto 1 tendremos:

```
1 | 8.00000 | 0.00000 | 0.00000 | 0.113321e+01 | -.132312e+01 | 0.962336e-02
|-.179831e+00 | -.362833e-01 | 0.520126e+00 |
```

Recordando lo visto en el apartado dedicado a la normalización de los ficheros, vemos que la columnas son:

```
Id Punto | X | Y | Z | ReX | Imy | ReY | ImY | ReZ | ImZ |
```

Teniendo en cuenta que:

$$Mod = \sqrt{Re_X^2 + Im_X^2}$$

resulta

$$Mod_X \approx 1{,}742071 \text{ m} \quad : \quad Mod_Y \approx 0{,}180088 \text{ m} \quad : \quad Mod_Z \approx 0{,}521390 \text{ m}$$

Para una frecuencia de 19,03 rad/s y una velocidad de 24 fotogramas por segundos, tenemos:

$$T_T = 1/24 \approx 4{,}166 \cdot 10^{-2} \text{ fp/s} \quad : \quad N_F = 1 \text{ fp}$$

$$t = T_T \cdot N_F = 4{,}166 \cdot 10^{-2} \text{ s}$$

Por tanto, la posición x, y, z para el primer cuadro (frame 1) se obtendría de forma:

$$\Delta X = 1{,}742071 \cdot \cos\left(19{,}03 \cdot 4{,}166 \cdot 10^{-2}\right) \approx 1{,}22269 \text{ m}$$

$$\Delta Y = 0{,}180088 \cdot \cos\left(19{,}03 \cdot 4{,}166 \cdot 10^{-2}\right) \approx 0{,}12639 \text{ m}$$

$$\Delta Z = 0{,}521390 \cdot \cos\left(19{,}03 \cdot 4{,}166 \cdot 10^{-2}\right) \approx 0{,}36594 \text{ m}$$

Con estos resultados conocemos que la variación de la posición del nodo 1, es decir, cuanto varía la posición del punto 1 en torno a su posición estática de equilibro indeformado (posición inicial antes de aplicarse la deformación). Esta información se encuentra en las tres primeras columnas del punto en el fichero movimiento. En este caso, la posición inicial es:

$$X = 8{,}0000 \quad ; \quad Y = 0{,}00000 \quad ; \quad Z = 0{,}00000$$

Por tanto la posición final del nodo en el fotograma 1 es:

$$X + \Delta X = 8 + 1{,}22269 \approx 9{,}22269 \text{ m}$$

$$Y + \Delta Y = 0 + 0{,}12639 \approx 0{,}12639 \text{ m}$$

$$Z + \Delta Z = 0 + 0{,}36594 = 0{,}36594 \text{ m}$$

El proceso se realizará de la misma forma para cada uno de los nodos de la malla. Cuando se proceda con el último vértice, estaremos en disposición de representar la malla deformada en el fotograma 1.

Si procedemos de la misma manera para cada fotograma del intervalo de tiempo marcado para la animación, resultará la sucesión de posiciones deformadas en cada instante de tiempo necesario para representar el movimiento completo del modelo poligonal.

8. Escritura del Cache

Una vez comentados los principios teóricos y técnicos de la información cache, solo queda explicar como se escriben los datos.

Elesys realiza, en primer lugar, la recolección de los datos del fichero importado. La información es almacenada en una matriz intermedia para su posterior uso. Además, toma los parámetros introducidos por el usuario e igualmente necesarios para la discretización del movimiento en fotogramas. Valores como número de fotogramas por segundos, amplitud del movimiento, velocidad de la animación y otras variables que estarán disponibles en la interfaz gráfica, para que sean determinadas por el usuario, como se explicará en capítulos posteriores.

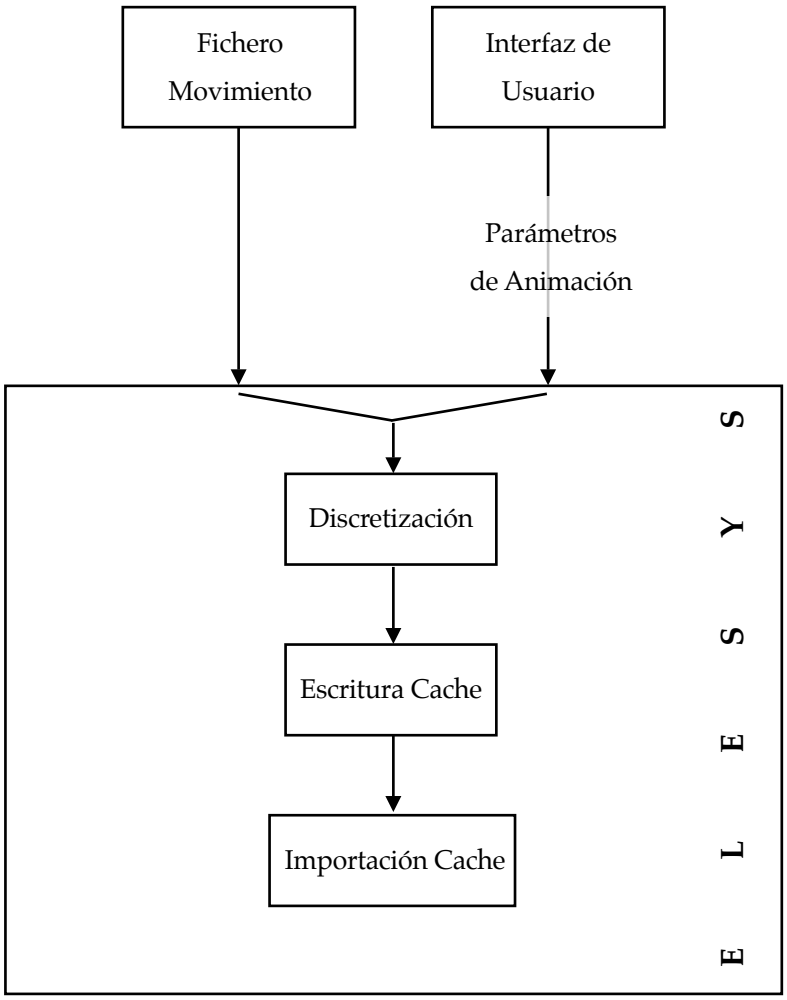


Figura 8.14. Esquema del proceso de animación del modelo poligonal

Es importante que la recolección de datos sea de manera rápida, ordenada y efectiva. Se ha huido de recorridos múltiples por el fichero de datos, indexación problemática y otros aspectos que resten eficiencia y dinamismo al proceso.

Una vez importados los datos procedemos con las tareas internas que realiza Elesys. Aunque se ha realizado un diagrama lineal en la figura 8.14., la discretización y escritura del cache son proceso realizados en paralelo con el fin de no utilizar bloques de datos intermedios, duplicar los procesos de lectura/ escritura, etc. El proceso en paralelo permite un realización directa del cálculo y escritura acelerando y optimizando los resultados.

Hemos explicado anteriormente como obtener la posición deformada de cada nodo en cada uno de los cuadros de la animación. Pues bien, cada vez que obtenemos el resultado de un nodo, este se almacena escribiéndose en un fichero cache intermedio. Elesys crea ficheros binarios que funciona como almacén de datos de resultados que posteriormente serán volcados en el cache.

Podríamos preguntarnos si no sería más idóneo volcar los resultados directamente en la memoria cache sin este paso intermedio. Esta metodología tiene una serie de ventajas enumeradas a continuación:

- El fichero intermedio nos permite tener siempre a disposición de Elesys los datos ya procesados, con el fin de poder ser reutilizado posteriormente sin tener que realizar nuevamente los cálculos.

- De esta forma se protege al *vertex cache* de posibles desbordes ante modelados polígonales con gran número de nodos.

- No se "esclaviza" la cache con Elesys, sino que la caché puede seguir trabajando con las demás aplicaciones y procesos de la computadora que pudiera necesitarla.

9. Composición de Dependency Nodes en el DG

Como ya se comentó, todas las acciones realizadas en Maya se plasman en el DG. Cuando Elesys realiza las tareas correspondientes a la animación, en el plano más esencial, está creando nodos de dependencia y conexiones entre ellos para que exista un correcto flujo de datos en el DG.

A continuación se comentará que procesos realiza Elesys en esta capa más básica de Maya, pudiéndose comprender que es lo que ocurre realmente con los datos de la animación. La exposición se realizará desde un punto de vista descriptivo, obviando los detalles más técnicos que no aporten interés.

Inicialmente, después de la generación del modelo, el DG se encuentra compuesto por:

- Un nodo superficie PolySurface1, que contiene la información de la malla generada.
- Un nodo de sombreado initialShadingGroup, que aporta los datos necesarios para que Maya sepa que tipo de sombreado debe aplicar al objeto y cual debe ser su comportamiento ante la luz cuando se realice el proceso de render.

Esta disposición del DG puede verse en la figura 8.15.

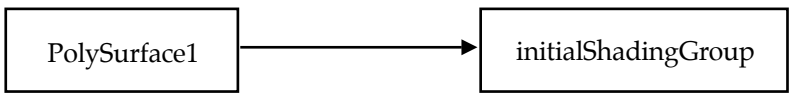


Figura 8.15. Esquema del DG después de la generación del mallado

Si a continuación realizamos el proceso de importación del movimiento, el DG se complica.

Al importar el fichero de movimiento, como se expuso en apartados anteriores, Elesys lee los datos que contiene. Posteriormente, toma los parámetros relacionados con la animación introducidos por el usuario en GUI. Con todo ello comienza a discretizar el tiempo y escribe los datos de caché.

Realmente, estos datos se escriben en el disco duro. Estos datos son escritos en formato binario y la discretización de cada fotograma es almacenada en un fichero distinto. Esto permite que el trasvase de información entre el disco y la caché de la GPU sea lo suficientemente rápida para que no se produzcan tiempos de espera.

Por otro lado, en el DG de Maya se generan los siguientes nodos de dependencia:

- polySurfaceShapeCache: contiene en sus atributos los parámetros de configuración de la caché de geometría. Es decir, no contiene la posición de los puntos del mallado sino datos como: ruta del fichero de caché, fotograma de inicio de la animación, fotograma de finalización de la animación, etc.

  Este nodo de dependencia permite a Maya saber como cargar, donde localizar y cuando hacer uso de los datos del caché.

- cacheBlend: este nodo sí contendrá los datos propiamente dichos del caché. Se compone principalmente de dos atributos:

  - outCacheData: array que contendrá la posición de cada uno de los vértices que compone la figura.

  - inRange: variable booleana que indicará a Maya si el fotograma en el que se encuentra, esta bajo el rango de la geometría caché o no. Si lo está, buscará la posición de los vértices en ese instante y aplicará la deformación. Si esta fuera de rango, no aplicará ningún tipo de modificación sobre la malla inicial.

- cacheSwitch: es el puente de conexión entre las variables double de el caché de geometría (que recibe como entrada) y la geometría propiamente dicha (atributo de salida). Permite, por tanto, conectar datos de entrada de una cadena de datos y da como salida la geometría ya deformada.

Tan importante como el número y función de cada nodo de dependencia es la conexiones que estos tienen dentro del DG. La figura 8.16. muestra los nodos de dependencia y sus conexiones, tal y como quedarían en el DG.

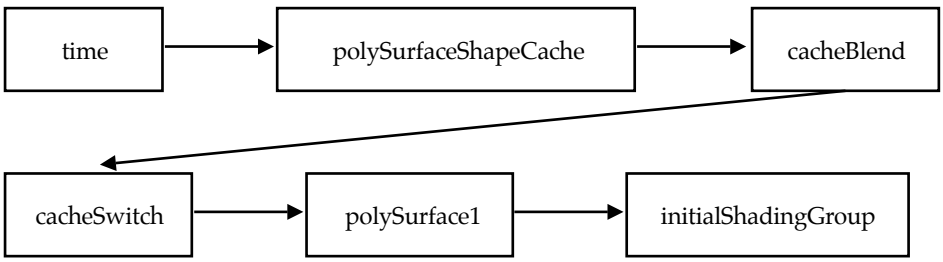


Figura 8.16. DG en Maya después de generar el movimiento

Vemos como el nodo time conecta con el polySurfaceShapeCache, lo que permite a este último, ser consciente de las variaciones en el tiempo, como por ejemplo, cuando se mueva el *slider* o deslizador en la interfaz de Maya o cuando se pulse el botón *Play* para visualizar la animación en los visores.

Esto permite que se produzca la actualización de los datos en los nodos de dependencia cada vez que se modifique el tiempo. Así, cuando el tiempo varía 24 veces por segundo (PAL 24 f/s), todos los nodos involucrados en el cache se actualizarán al mismo ritmo, produciéndose la animación en tiempo real en los visores.

Siguiendo la línea de conexiones, encontramos al nodo cacheBlend que está conectado a polySurfaceShapeCache. El nodo polySurfaceShapeCache aporta mediante esta unión los parámetros de la animación que permite al cacheBlend tomar los datos del caché y almacenarlos en sus atributos. Por ejemplo, el

polySurfaceShapeCache da la ruta del fichero de datos almacenado en el disco. El cacheBlend abrirá ese fichero y almacenará los datos concernientes al mismo.

Recordemos que la posición de cada vértice en cada fotograma se almacena en un fichero distinto fotograma. Así, la lectura y almacenamiento de datos del cacheBlend será continua y constante mientras cambie el tiempo.

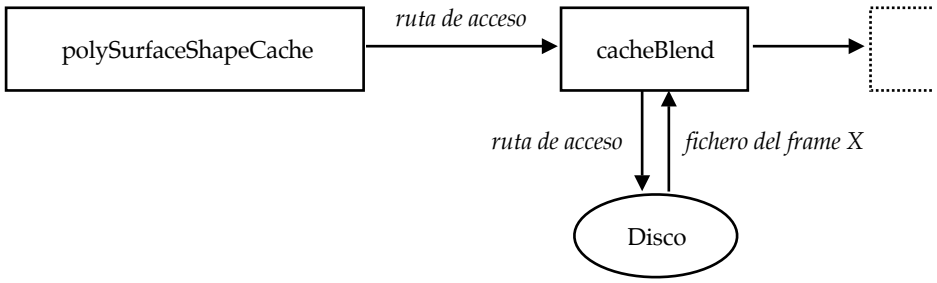


Figura 8.17. Detalle del intercambio de información del nodo cacheBlend

El nodo de cacheSwitch sirve de unión entre los datos de entrada (cadena de las posiciones de cada vértice, outCacheData[]) con la salida (geometría deformada, outputGeometry). Este nodo importa la cadena de coordenadas de los vértices y aplica la deformación a la malla. De esta forma, tienen como salida la geometría deformada que posteriormente será tomada como entrada por el nodo polySurface, nodo de mallado que es representado en el visor.

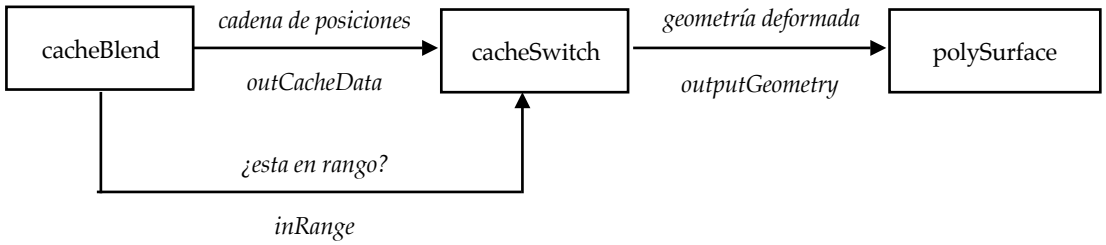


Figura 8.18. Detalle de conexiones del nodo cacheSwitch

El nodo cacheBlend también proporciona una conexión inRange al cacheSwitch. Consiste en una variable lógica, que será verdadera cuando el fotograma actual esté dentro del rango del cache de geometría y falsa cuando esté fuera del rango.

Cuando esta variable sea falsa la geometría proporcionada al polySurface será la malla sin deformación.

La composición del DG aquí expuesta, esta particularizada para los nodos, atributos y conexiones que Elesys crea para proporcionar movimiento al objeto. A medida que se le realice operaciones suplementarias, el DG irá añadiendo y eliminando nodos de dependencia según necesite.

Los detalles del DG para otras operaciones como simetrías, suavizado, color por vértices, etc. serán analizadas en los capítulos correspondientes a cada uno de estos temas.

Edición del Modelo

Los resultados que Elesys importa para su procesado son datos de fuentes externas. El origen de esos datos está en un complejo proceso de cálculo mediante el método de elementos finitos o elementos de contorno.

Por muy bien que se halla planeado la etapa de preprocesado, puede ocurrir que sea necesario procesos de edición de los datos originales con el fin de un mejor análisis. Los motivos de la edición del modelo original pueden ser de origen muy diverso. Puede deberse desde la propia configuración geométrica del modelo hasta por la obtención de resultados no previstos inicialmente en la etapa de preprocesado.

Sea como sea, Elesys debe permitir que el usuario controle ciertos aspectos de la representación para que pueda conseguir mayor calidad en las tareas de postprocesado.

Este capítulo describe detalladamente las herramientas diseñadas en Elesys para el desempeño de este objetivo. A través de él, se presentará cada uno de los procesos y se comentarán qué razonamientos y planteamientos se ha realizado para cumplir los objetivos de diseño marcados previamente.

1. Objetivos de la Edición

Como se ha comentado, las necesidades de edición de un modelo puede tener orígenes muy distintos. La implementación de un número excesivo de filtros y tareas de edición, obligaría un proceso de aprendizaje de los distintos parámetros que iría en contra de la filosofía de simplicidad y facilidad de uso marcada en las primeras etapas de diseño de Elesys.

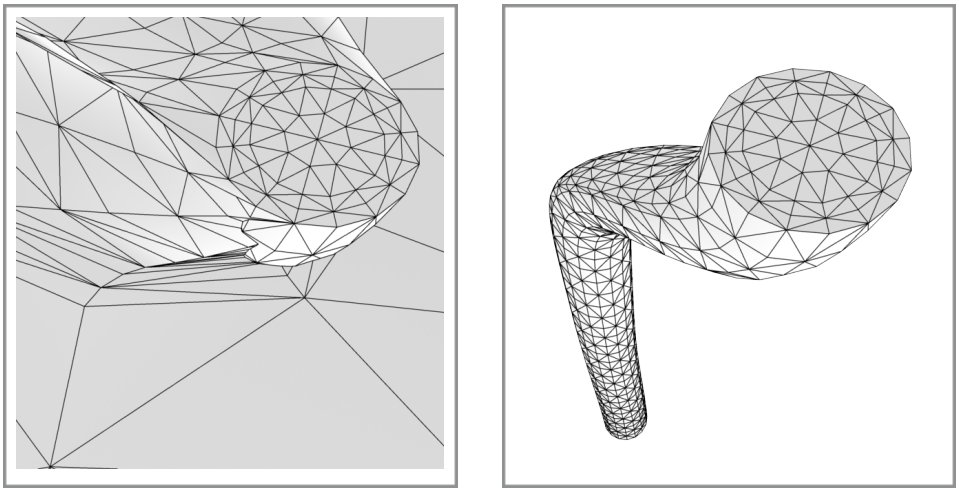
Con el fin de preservar esta filosofía, se ha decidido implementar sólo los procesos de edición más importantes. Además esta implementación deberá ser de la manera más sencilla e intuitiva posible.

De todos los procesos de edición existentes, a continuación pasaremos a destacar los más útiles que se ha decidido implementar. En este apartado se realizará una exposición general, con el fin de exponer su utilidad, desarrollando detalladamente en apartados posteriores los detalle técnicos.

Presentación detallada de un aspecto concreto del modelo.

En ocasiones puede presentarse la necesidad de analizar con mayor detalle una zona o aspecto en particular del modelo. En estos casos es preferible desechar aquellos aspectos que obstaculicen en la visualización del problema particular.

En la figura 9.1. se tiene un pilote sometido a una serie de fuerzas dinámicas. En este ejemplo, estamos ante un modelo de grandes dimensiones en el que se quiere observar la deformación del pilote desde un ángulo superior. Ésta, por la propia geometría del modelo, presenta una difícil visualización de los elementos en cuestión.



(A) (B)

Figura 9.1. En la figura superior (a) se observa como la capa superior no deja visualizar todo el pilote. Esto se consigue procediendo a un proceso de ocultación de ciertos elementos como se ver el resultado de la figura (b).

Proceso de suavizado de la malla.

En la etapa de preprocesado se concreta el modelo a estudiar. El número de nodos y elementos, así como su posición en el espacio, se define en este punto.

Es obvio que a mayor número de nodos, el modelo será más exacto y su comportamiento se acercará con mayor precisión al que se produce en el mundo real. Igualmente, la probabilidad de detectar posibles errores, anormalidades y otros efectos deseados o no deseados, será mayor cuanto más detalle tenga la malla.

El aumento en el número de nodos tiene, sin embargo, sus efectos negativos. El principal y más evidente es el de consumo de potencia de cálculo. A medida que complicamos el modelo se añaden nuevas variables al problema, que necesitarán de mayor potencia de cálculo y tiempo para su resolución. Esto hace que sea necesario llegar a un compromiso entre detalle, tiempo y recursos de cálculo.

En la etapa de postprocesado puede resultar útil añadir nodos a la malla que permita refinar el comportamiento de ésta en el proceso de representación.

Aunque el comportamiento de los nodos añadidos no pueden argumentarme mediante resultados matemáticos estrictos, su evolución responde a interpolaciones geométricas de los nodos originales del modelo. Este método de refinamiento puede resultar muy útil proporcionando mayor suavidad en la malla con un coste no muy alto de recursos de computación.

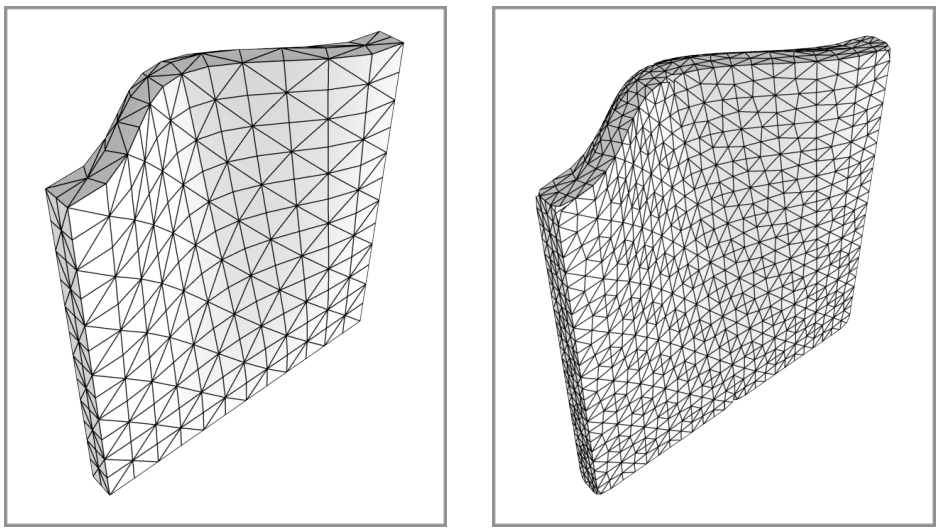


Figura 9.2. La malla izquierda muestra el número de nodos original.
La de la derecha es el resultado de aplicar un filtro de suavizado en la malla original.

Simetrías y Antisimetrías.

Por los motivos comentados anteriormente, es obvio que, el consumo de recursos de computación que se realiza en un problema de elementos finitos o de contorno está íntimamente ligado al número de nodos que tiene la malla.

Los casos de elementos con geometría y comportamiento simétrico y/o antisimétricos resultan especialmente atractivo en procesos de edición en la etapa de postprocesado. En estos casos no es necesario el planteamiento, cálculo y resolución del modelo completo. Basta con resolver la parte del problema que tiene planos de simetría y/o antisimetría y posteriormente, mediante procesos de edición, representar las partes omitidas.

El ahorro de tiempo y recursos es muy importantes, de aquí, el especial interés de estos casos. La figura 9.3. muestra una figura que tiene un plano de simetría con respecto al eje X y un plano de antisimetría con respecto al eje Y.

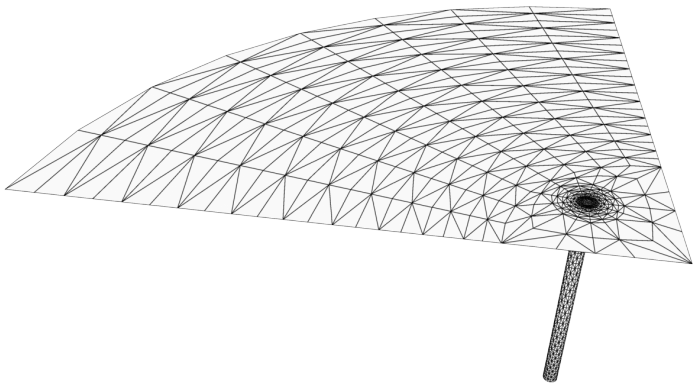


Figura 9.3.a. Modelo definido para resolución. Se estudia un cuarto del modelo completo.

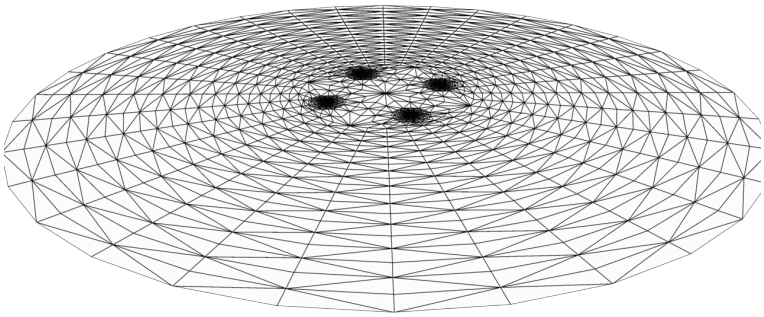


Figura 9.3.b. Modelo completo obtenido mediante proceso de edición de simetría en el eje X y antisimetría en el eje Y.

En este caso basta con resolver una cuarta parte del problema, y posteriormente, representar la parte simétrica y antisimétrica del modelo. En este caso sólo se necesitó un 25% de los recursos que se hubiera necesitado en el caso de haberse planteado y resuelto el modelo completo.

2. Ocultación de Elementos

La ocultación de elementos de una malla es un método muy útil a la hora de representar zonas del modelo que por alguna razón no son visibles fácilmente. También ayuda a la hora de realizar secciones o cortes de la figura por aquellos planos que tenga un especial interés dentro del proceso a estudiar.

Desde el punto de vista del desarrollador, el proceso tiene una primera decisión que analizar. Se puede plantear la ocultación de elementos como tal, o bien la eliminación de los elementos que no se quieran mostrar. La supresión de las partes no necesarias de la figura facilitan el proceso en un primer momento. Sin embargo no puede afirmarse lo mismo en el caso que se de la opción al usuario de restaurar el modelo, volviéndose a mostrar los elementos eliminados. Esto obligaría a crear de nuevo lo eliminado y aplicar aquellas modificaciones, filtros y procesos utilizados en el resto del modelo no eliminado. Por ello esta opción es desechada en pro de la implementación de un algoritmo que verdaderamente oculte la malla sin eliminarla.

El proceso se implementa en Elesys separándolo en dos partes diferenciadas:

- Selección inteligente de los componentes a esconder.
- Proceso de ocultación propiamente dicho.

La selección se diseña de forma que pueda realizarse de forma manual (por el usuario) o de forma automática (por la computadora). El modo manual permite que la persona elija los elementos, ayudado por la facilidad del entorno de visualización de Maya. Mediante giros de cámara que se realizan fácilmente con el ratón, el usuario puede colocar la vista del modo más conveniente para, por último, proceder a la selección mediante sucesivos clics sobre los elementos triangulares o la selección múltiple mediante un cuadro de selección.

El modo automático funciona de forma muy diferente. El usuario, mediante la interfaz de usuario que proporciona Elesys, puede dar una serie de parámetros espaciales que permitan posteriormente a la computadora, elegir de forma inteligente aquellos elementos que cumplan con los parámetros aportados.

Con el fin de proporcionar la facilidad de uso y la simplicidad buscada durante todo el proceso de desarrollo de Elesys, las reglas de decisión aportadas al sistema no pueden ser complejas ni confusas. Se decide así, que el aspecto más esencial de un elemento para su ocultación es su posición en el espacio. Por ello, las variables que intervienen en la selección son unas fáciles reglas de posición.

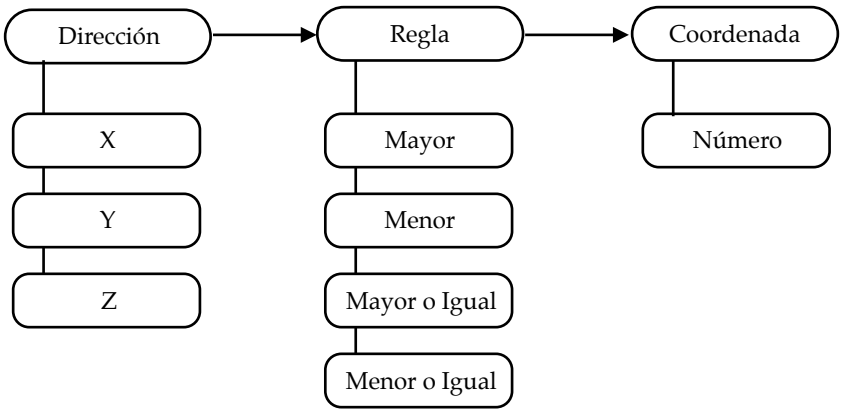


Figura 9.4. Esquema de reglas de decisión de elementos a ocultar

La figura 9.4. indica las posibilidades que Elesys da para la selección automática. Como se observa, a través de esta regla de decisión, el sistema seleccionará aquella parte de la malla que cumplan con estas indicaciones referidas a su posición en el espacio. De esta forma, se puede realizar secciones muy rápidamente (figura 9.5) y ocultar partes poco interesantes para la visualización de un fenómeno.

Figura 9.5. Sección de un muro con parámetros $Z \geq 75.0$.

Hay que tener en cuenta que la selección debe realizarse desde el punto de vista de los vértices de caras. Sin embargo, la evaluación de los parámetros se realiza en los nodos. De esa forma, con que uno de los nodos que conforma una cara cumpla la condición marcada por el usuario, será ocultadas (o mostradas) todas las caras de las que forma parte.

Para fijar ideas, se concretará con un ejemplo. Supongamos que tenemos la cara o elemento formado por tres nodos tal y como muestra la figura 9.6.

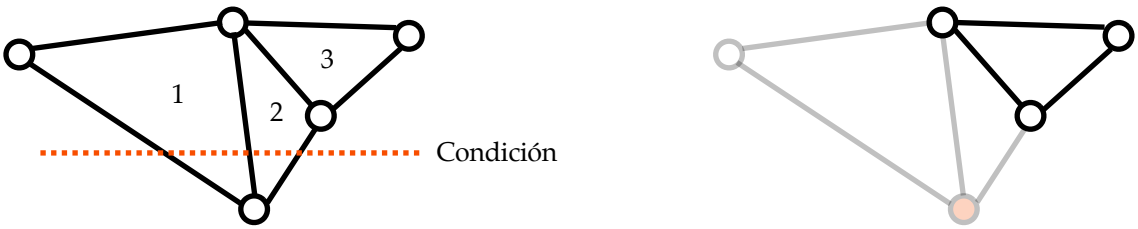


Figura 9.6. En la parte izquierda se muestra tres elementos con cinco nodos.
En la parte derecha el resultado de aplicar una ocultación con la condición indicada.

Vemos en la parte izquierda cómo se tiene tres elementos (numerados del 1 al 3) con cinco nodos. La línea punteada indica la condición impuesta. Con esta premisa, sólo el nodo inferior es seleccionado. Como los elementos 1 y 2 están formados con este nodo. En este caso, al ocultar este vértice, se debe esconder las

dos caras (1 y 2). Así la parte derecha de la figura muestra el resultado de la operación.

3. Suavizado de Elementos

En ocasiones, la malla poligonal no aparece con la apariencia suavizada que se quisiera. Las causas pueden ser muchas, desde una definición del problema con escaso detalle hasta un movimiento de gran amplitud entre nodos contiguos.

Cuando esto ocurre, existen varios métodos de suavizado que pueden aplicarse, con el fin de obtener aproximaciones del modelo, que permitan una visualización más refinada del problema.

Dentro de los métodos de suavizado podemos dividirlos en dos partes:

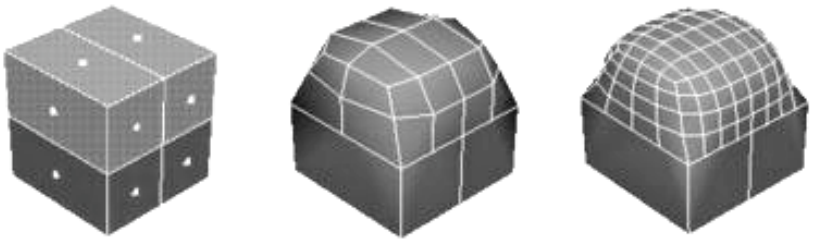
- Suavizado exponencial.
- Suavizado lineal.

Ambos métodos tienen opciones de suavizado similares pero ofrecen distintos métodos de control para la topología resultante. Por ejemplo, el exponencial tiene una opción para mantener invariables los bordes del mallado mientras que la opción lineal permite un mejor control del número de caras resultantes.

Control Exponencial

Cuando se hace uso del suavizado exponencial, están disponibles las siguientes opciones:

- Nivel de División: indica las veces que Maya aplica el proceso de suavizado sobre el mallado. Esto incrementará o disminuirá el nivel de suavizado. El rango del nivel de división es de 1 a 4. El valor mayor indica el objeto más suavizado.



Malla Original Nivel de División 1 Nivel de División 2

- Continuidad: El valor que se introduce determina el grado de suavizado.

Continuidad 0,0 Continuidad 0,2 Continuidad 0,5

- Suavizado UVs: Aplica la misma operación de suavizado a los vértices UVs. El suavizado UVs está seleccionado por defecto, proporcionando unos mejores resultados para los UVs.

- Propagación de los límites: copia los límite de la malla original a los nuevos límites asociados a la malla suavizada. Por defecto, esta opción está desactivada.

- Mapeado de bordes: controla cómo son suavizados los bordes cuando el suavizado UVs está activado. Las opciones disponibles son: no suavizar, suavizar internamente, suavizar todo.

- Preservar: especifica qué componentes permanecerán sin afectar durante el suavizado. Las opciones son:

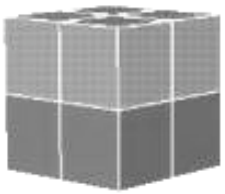
Antes Suavizado

Geometría de bordes

Sin Geometría de Bordes

Geometría de bordes: cuando está activado, esta opción preserva las propiedades de los límites de bordes de la malla.

- Bordes seleccionados: esta opción conserva las propiedades de los límites de bordes seleccionados y caras no seleccionadas.
- Bordes duros: conserva las propiedades de cualquier borde.



Antes Suavizado

Selección de bordes

Sin Selección de Bordes

Controles Lineales.

El suavizado lineal da mayor control sobre el número de caras resultantes del suavizado, especialmente a través del atributo de divisiones por cara.

Este método de suavizado no debe usarse cuando se tiene más de cuatro bordes, ya que el resultado sería una superficie suavizada desigual. Sin embargo, las mallas generadas por Elesys están compuestas por polígonos triangulares por lo que este problema no nos afectará.

Las opciones disponibles para el manejo del suavizado será:

- Nivel de división: Es el número de veces que maya aplica el proceso de suavizado sobre el modelo. A mayor número, se generará una superficie

más suavizada y con mayor número de caras. Sin embargo esta opción hay que tomarla con precaución puesto que aumentando este número, fácilmente se puede generar un modelo con un número excesivo de caras, que puede reducir la interactividad de Elesys con el usuario. Esto puede dar problemas a la hora de representar en el visor animaciones y otros fenómenos en tiempo real.

- División por cara: Aumenta el número de caras con un menor incremento que el nivel de división. Usando esta opción, puede conseguir con más facilidad un mejor compromiso entre un balance apropiado de suavizado y bajo número de polígonos. El número de polígonos aumenta a consecuencia de dividir cada una de las cara originales. Elesys hace la división partiendo los bordes existentes. El valor que utilizado en este parámetro indica el número de separaciones Elesys realiza. Cuándo se coloca una división, cada borde es dividido una vez, un valor de 2 lo dividirá dos veces, etc.

- Fuerza de empuje: controla el volumen total de la malla suavizada resultante. Valores más altos escalan la malla hacia afuera mientras que, valores más bajos lo escala hacia dentro. La configuración del valor por defecto es 1.

- Redondez: crea un efecto de bombeo en la superficie a través de un escalado de vértices alrededor del centro de las caras originales. Los valores más altos escalan estas cimas hacia afuera y valores más bajos lo escalan hacia dentro. Para que la redondez tenga efecto, la fuerza de empuje debe ser mayor de cero.

Solución adoptada.

Siguiendo con la filosofía del proyecto de simplicidad y facilidad de uso, no se ha querido dejar en manos del usuario la complicada tarea de decidir en cada

momento los parámetros de control más apropiado para el modelo utilizado. En su lugar, se ha querido simplificar esta decisión proporcionando una serie de valores predefinidos que aseguren los mejores resultados para el tipo de mallas estudiadas.

Después de innumerables test de forma, efectos sobre el movimiento, representación de nodos y otras pruebas adicionales se ha podido estudiar los puntos fuertes y débiles de cada uno de los suavizados así como las mejores combinaciones en los valores de los parámetros de control de cada uno de los métodos.

De esta forma, se ha adoptado finalmente el método de suavizado exponencial. Se comprobó que este método tiene mejores resultados en movimientos por vértices como el que Elesys maneja, respetando mejor la exactitud y rigor de los datos.

En primer lugar, el control sobre la forma de la malla es mucho mayor. Se consigue así, que no se produzcan efectos no deseados, especialmente en los encuentro de caras con ángulos menores a 90º, conservándose los bordes y límites de caras de forma apropiada. Igualmente, se consigue una mejor conservación de los nodos originales, produciéndose el suavizado alrededor de estos. Todo ello, gracias a la conservación de la geometría de los bordes y niveles de división.

Así, se pone a disposición del usuario, tal y como se verá posteriormente en el capítulo de la interfaz de usuario, un sencillo deslizador que indicará el grado de suavizado a aplicar a la malla. En función de este número, Elesys aplicará la mejor combinación de parámetros de control exponencial teniendo como resultado una malla refinada, lisa, sin perder el grado de rigor de los resultados aportados inicialmente.

Desde el punto de vista del DG no hay grandes cambios. Simplemente, se añade el nodo polySmoothFace, que tendrá como entrada la malla sobre la que se va aplicar el suavizado y como salida, esa malla con el proceso aplicado.

9.7. Representación del suavizado en el DG.

4. Simetría / Antisimetría

Como explicamos anteriormente, la simetría puede aportar gran cantidad de tiempo ahorrado en computación y cálculo, teniendo en cuenta que los resultados obtenidos para la parte original de una figura podrán aplicarse, igualmente, a su parte simétrica (o antisimétrica).

El responsable de las tareas de preprocesado debe cerciorarse que el problema es simétrico en todas y cada una de las variables a calcular. De no ser así, es evidente que los resultados aplicados a la parte simétrica o antisimétrica no podrán ser válidos.

Elesys toma como figura original o base, a aquel trozo de la malla que mediante sucesivas operaciones de simetrías y/o antisimetrías pueden originar la figura completa. De esta forma, será sobre esta figura base, sobre la que se aplicarán los ficheros de resultados obtenidos de la fase de cálculo. Una vez importados los ficheros de cálculo, será tarea de Elesys aplicar correctamente una extensión de estos resultados a las partes añadidas a la figura.

Con el fin de mantener la simplicidad del programa y su facilidad de uso, Elesys sólo necesitará dos parámetros para realizar la operación: tipo de operación y eje en el que aplicar dicho proceso. El tipo de operaciones que se le puede indicar a Elesys son, evidentemente, dos: simetría o antisimetría. Los ejes en los que puede aplicar la simetría son: ±x, ±y, ±z.

Con ello, se evita al usuario, el realizar ningún tipo de cálculo previo para obtener el plano de simetría y no es necesario dar a Elesys complicados parámetros de direcciones, ecuaciones de planos, etc. Elesys lleva implícito en la representación de la simetría el cálculo de plano a partir del que se aplica la operación. Supongamos que aplicamos una simetría a una figura en el eje +X (figura 9.7.).

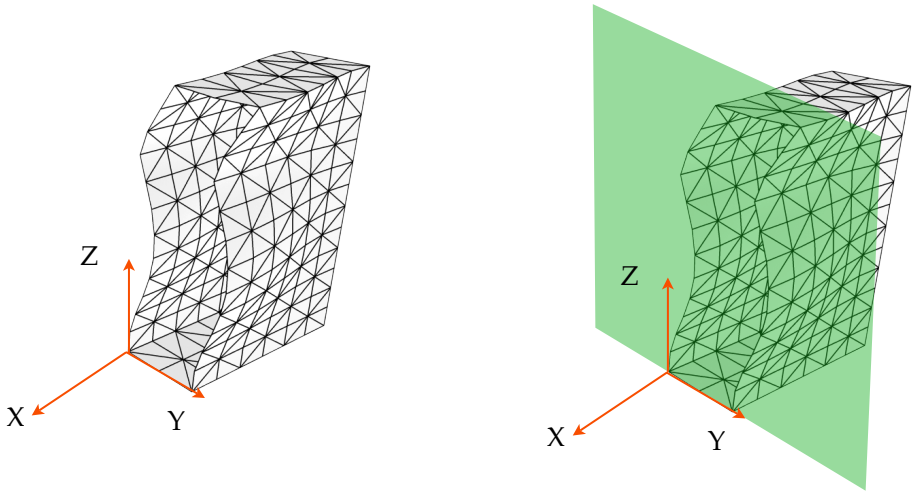


Figura 9.8. En la parte izquierda, figura base para aplicar simetría en el eje +X. En la parte derecha, plano de simetría calculado.

Para ello la aplicación procede de la siguiente forma:

- Determina cuál es el punto más alejado del origen en el eje X positivo.
- Coloca en ese punto el plano de simetría perpendicular al eje.
- Aplica la simetría propiamente dicha tomando el plano del punto anterior (Figura 9.8.).

En la figura 9.7. se observa la malla original. Al aplicarse la simetría en el eje +X, se buscará el nodo con la coordenada X mayor. En este caso, la figura se extiende hacia el semieje negativo, por ello, la coordenada mayor (o menos negativa) es

X = 0. Por tanto, el plano de simetría será perpendicular al eje X en el punto X=0 como se ve en la figura AA en la parte derecha.

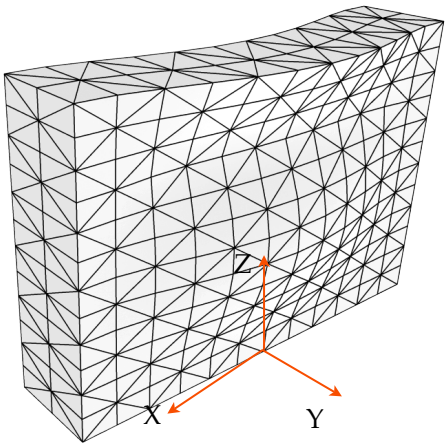


Figura 9.9. Resultado de aplicación de la simetría en el eje +X.

Así pues, ya sólo queda aplicar la simetría propiamente dicha. La figura 9.8. muestra el resultado final. Este caso, ya explicado, puede aplicarse igualmente al caso de la antisimetría.

Desde el punto de vista de el DG, Elesys realiza este proceso de simetría o antisimetría creando los nodos polymirror o polyantisim. Estos nodos funcionan de forma distinta a pesar de producir efectos similares.

El nodo polymirror almacena la geometría original del cuerpo a procesar en su atributo de entrada inputPolymesh.

También incorpora como entrada la matriz de transformación de la malla resultante worldMatrix. Las matrices en Maya son el significado de la combinación de una serie de transformaciones. Esta transformaciones pueden combinarse mediante el proceso denominado **concatenación**. Así, el nodo polymirror puede conocer las transformaciones que ha sufrido la malla original y concatenar su propia transformación que le permite obtener la figura simétrica.

También podemos encontrarnos con otros atributos que configuran la realización de la simetría. Estos son parámetros internos al nodo, como es el punto pivote de la transformación (pivot), la posibilidad o no de fusionar ambas figuras (merge) o la dirección de la simetría (direction).

El nodo polymirror aporta una geometría de salida con la figura original con su correspondiente figura geométrica. Esta geometría se encuentra en su atributo output.

En la figura 9.10. podemos ver el nodo "a" con sus respectivos atributos y conexiones:

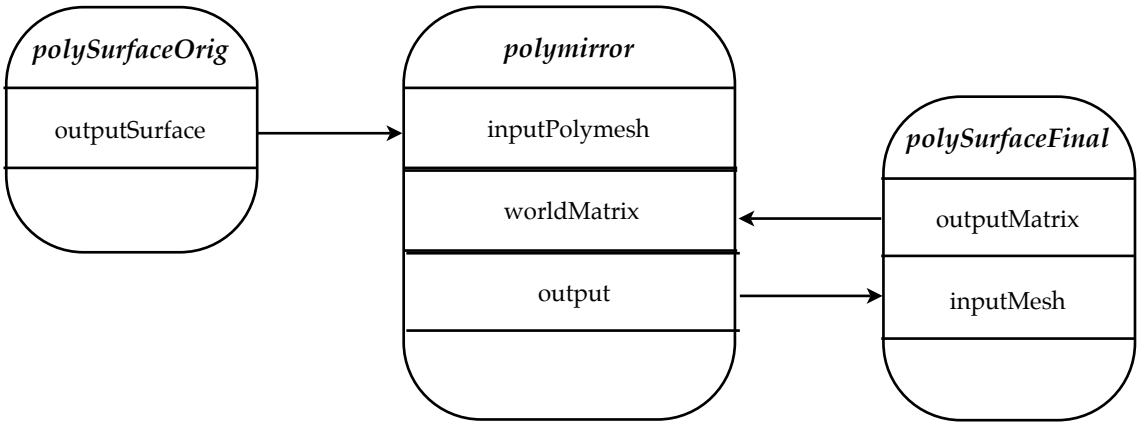


Figura 9.10. Nodo polymirror y relaciones con otros nodos del DG

Para la antisimetría se ha programado un nodo que le permita a Elesys realizar este proceso. El nodo se denomina polyantisim y cuenta con los siguientes atributos:

- InputSurfaceOrig: este atributo importa la versión más primitiva de la malla antes de que sea modificada por otros nodos del DG. Sirve para capturar la geometría de los puntos sin deformar, con el fin de tener la posición de cada vértice de los que posteriormente será generado su correspondiente punto antisimétrico.

Sirve como mero recolector de datos de la malla original, puesto que la función de computación del nodo necesita esta información original. Por ejemplo, el número de vértices de la malla no sería correcto si se calculara después de un proceso de suavizado.

- InputSurface: al contrario del atributo anterior, éste importa la versión final de la malla. Sobre esta geometría será sobre la que realmente se aplicará la antisimetría. De esta forma, la mitad generada conservará todas las transformaciones aplicadas sobre la mitad original.

- OutputSurface: este atributo de salida enviará al siguiente nodo la malla total resultante de la aplicación de la antisimetría.

- Direction: mediante el uso de esta información se comunica al nodo hacia qué dirección debe aplicar la antisimetría. Este atributo se indicará en la interfaz de usuario (GUI) y no mediante conexión con otro nodo.

- Pivot: indica el punto que funciona como pivote de la copia a la mitad original. El valor por defecto será (0, 0, 0).

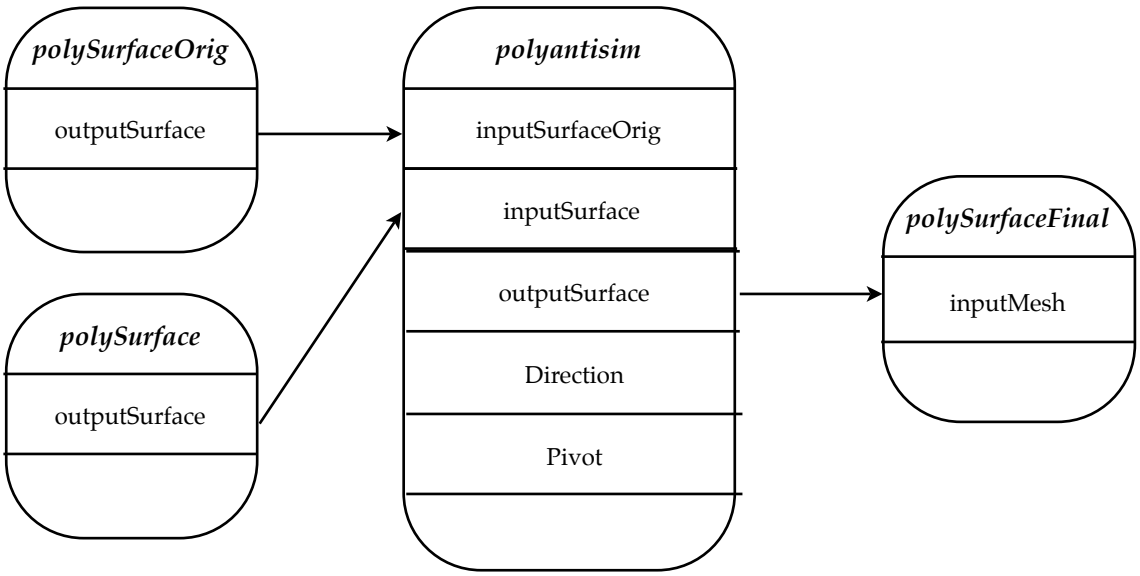


Figura 9.11. Representación esquemática de las relaciones del nodo polyantisim en el DG

Proceso de Renderizado

La renderización es el proceso de generar una imagen desde un modelo. La palabra renderización proviene del inglés *render*, y no existe un verbo con el mismo significado en español, por lo que es frecuente usar las expresiones renderizar o renderear.

En términos de visualizaciones en ordenador, más específicamente en 3D, la "renderización" es un proceso de cálculo complejo desarrollado por un ordenador destinado a generar una imagen 2D a partir de una escena 3D. La traducción más fidedigna de la palabra renderizar es "interpretación". Así podría decirse que en el proceso de renderización, la computadora "interpreta" la escena en tres dimensiones y la plasma en una imagen bidimensional.

En infografía este proceso se desarrolla con el fin de imitar un espacio 3D, formado por estructuras poligonales, comportamiento de luces, texturas, materiales (agua, madera, metal, plástico, tela, etc.) y animación, simulando ambientes y estructuras físicas verosímiles. Unas de la partes más importantes de los programas dedicados a la infografía son los motores de renderizado, los

cuales son capaces de realizar técnicas complejas como radiosidad, *raytracing* (trazador de rayos), canal alfa, reflexión, refracción o iluminación global.

Cuando se trabaja en un programa de diseño 3D por computadora, normalmente no es posible visualizar en tiempo real el acabado final deseado de una escena 3D compleja ya que esto requiere una potencia de cálculo demasiado elevada, por lo que se opta por crear el entorno 3D con una forma de visualización más simple y técnica, y luego generar el lento proceso de renderización para conseguir los resultados finales deseados. El tiempo de render depende en gran medida de los parámetros establecidos en los materiales y luces.

En la actualidad, la expresión fotorrealismo, se refiere a imágenes generadas por ordenador que modelan las propiedades ópticas y físicas del mundo real de forma tan acertada que, al contemplarlas, algunas personas jurarían que se trata de fotografías.

1. Motores de Render

En una producción infográfica, el motor de render es el encargado de realizar los cálculos y tareas de computación necesarias para obtener una imagen 2D de la escena 3D que se maneja. Podemos decir que los motores de render sean, quizá, la parte integrante de una producción infográfica que cuenta con mayor complejidad y avances tecnológicos.

Como se comentó en capítulos anteriores, el hecho de haber seleccionado el entorno Maya para la creación de la aplicación, permite que Elesys haga uso de los motores de render disponibles para Maya. Estos motores se encuentran en primera línea en cuanto a tecnología y calidad en el mercado de la imagen digital.

Si empezamos por el más básico, encontramos el motor *Maya Software* que es motor predefinido y tiene la calidad básica para capturar todos los elementos de que dispone la escena gracias al método de *raytracing* que incorpora. A través de

este, traza rayos desde las fuentes de luz que producen efectos de reflexión y refracción. La calidad de este motor, aunque buena, resulta demasiado elemental no explotando la verdadera potencia de render de la que se dispone en Maya.

El segundo motor que encontrarmos, *Maya Hardware*, utiliza el propio procesador de la tarjeta gráfica del ordenador, por lo que produce renderizaciones rápidas y a tiempo real en el mismo visor de trabajo, lo cual lo hace especialmente indicado en la representación de las simulaciones de escenas con partículas y efectos de partículas, juegos, filmaciones, etc. Sin embargo, se precisan tarjetas gráficas muy potentes generalmente disponibles en las estaciones de trabajos u ordenadores personales de alta gama. Puesto que Elesys no trabajará normalmente con sistemas de partículas, y que los objetivos previos marcan su uso en ordenadores sin grandes requerimientos de hardware, este motor es descartado.

Por último Maya incorpora el motor de render *Mental Ray*. *Mental Ray* es un programa de render desarrollado por Mental images en Berlin (Alemania). Este motor (realmente un plugin de otros programas de 3D), es uno de los pocos software desarrollados en Europa que compiten con los desarrollados en EEUU o Canadá.

El primer punto fuerte de *Mental Ray* es su integración con Maya. Esto hace que, pese a ser un plugin externo, ha llegado a integrarse como el motor de render más usado haciendo que Autodesk abandone el desarrollo de sus propios motores de render. Así, constituye una opción más de renderización incorporada al programa con lo que no es necesario adquirirla aparte haciendo un desembolso económico extra.

Mental ray tiene un motor bastante completo que calcula desde sombras simples hasta segmentadas, compilación fotónica emitida por luces que se conoce como "Photon Map" y su propio sistema de aceleración, que permiten hacer algunas

tareas en tiempos bastante rápidos para el nivel de fotorealismo que maneja. Esta versatilidad y eficiencia ha hecho que *Mental Ray* haya sido empleado en películas, juegos y comerciales.

Existen a su vez otros motores de render externos de enorme potencia. Sin embargo, a pesar de los buenos resultados que se obtienen con estos módulos externos, no se han sometido a estudio debido al alto coste económico que conllevan.

2. Definición de la Escena

En capítulos previos quedó expuesto como Elesys genera la malla. El fichero de datos de mallado nos proporciona los datos necesarios para una representación apropiada del modelo pudiéndose así realizarse las tareas de postprocesado. El usuario podía así visualizar en el visor la figura generada.

Sin embargo, si se quiere obtener imágenes renderizadas de una cierta calidad, no se puede representar únicamente la superficie generada a través del fichero de datos. Es necesario contextualizar la representación dentro de un entorno agradable y práctico que, además de facilitar el proceso de análisis, también permita tener imágenes y vídeos de calidad suficiente para ser usados en medios audiovisuales.

A este entorno lo llamaremos *Escena*. La Escena será, además de otros modelos poligonales secundarios que sitúen la simulación, todos aquellos elementos que interactúen en el proceso de renderizado tales como luces, materiales, sombreadores, etc.

3. Definición de Mallados Secundarios

El diseño del entorno del mallado se ha realizado conforme a unos objetivos previos. Los puntos más importantes puede resumirse en:

- **Neutralidad**: el entorno debe acompañar sin distraer del tema principal que es la figura importada.
- **Visión circular**: la variedad de modelos que Elesys puede importar obliga a continuos cambios de cámara por lo que el entorno debe permitir ángulos de visión de 360º.
- **Simplicidad**: el número de elementos de la escena debe ser el menor posible con el fin de no complicar la selección de elementos, ni provocar un aumento del tamaño del fichero resultante de forma excesiva.

Siguiendo estas premisas se ha construido un entorno compuesto por dos mallados. El primero lo compone un plano sobre el que se apoyarán los modelos. El segundo es una figura esférica que permite giros de cámara de 360º sin que deje de acompañar a las imágenes renderizadas. La figura inferior muestra una vista esquemática de ambos elementos.

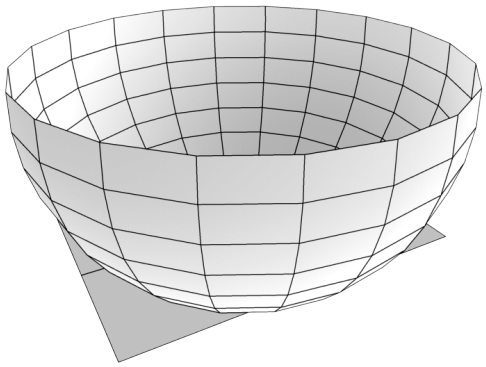


Figura 10.1. Vista general de la Escena

El color utilizado para la escena es el gris oscuro. El uso de este color neutro hace que los colores vivos del modelo contraste con el entorno y no distraiga al usuario de lo realmente importante, la malla importada.

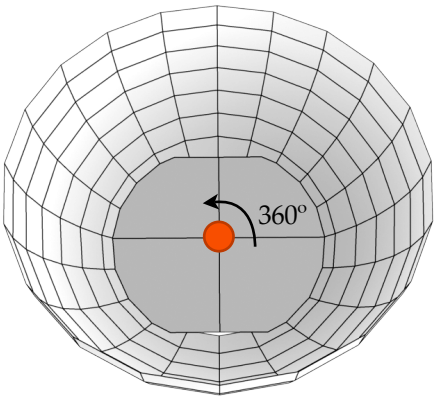


Figura 10.2. Vista Superior de la Escena.

En la figura 10.2. se indica con un punto la posición que ocupa la estructura importada. Desde ahí, se ve que los mallados secundarios acompañan como fondo, sea cual sea el punto de vista elegido por el usuario.

4. Modificaciones del Modelo

Elesys es un sistema de procesado en cuyo diseño no se ha puesto límites de modelos que puede estudiar. Todo sistema analizado bajo el método de elementos de contorno o elementos finitos puede ser importado en la aplicación.

Esta flexibilidad hace que Elesys pueda trabajar tanto con grandes estructuras de decenas de metros como con pequeñas piezas de un mecanismo. Así el usuario se verá obligado a cambiar el punto de vista de la cámara, acercarla o alejarla, etc.

Otra forma de adaptarse al problema de estudio consiste en modificar de forma relativa el modelo. La modificación relativa consiste en adaptar las características, posición, etc. del modelo, sin modificar en la misma medida los efectos producidos por las variables que actúan sobre él.

La figura 10.3. muestra una figura deformada a la que se le aplican distintos cambios de posición, rotación y tamaño, apreciándose como no se modifican las deformaciones que se aplican sobre él.

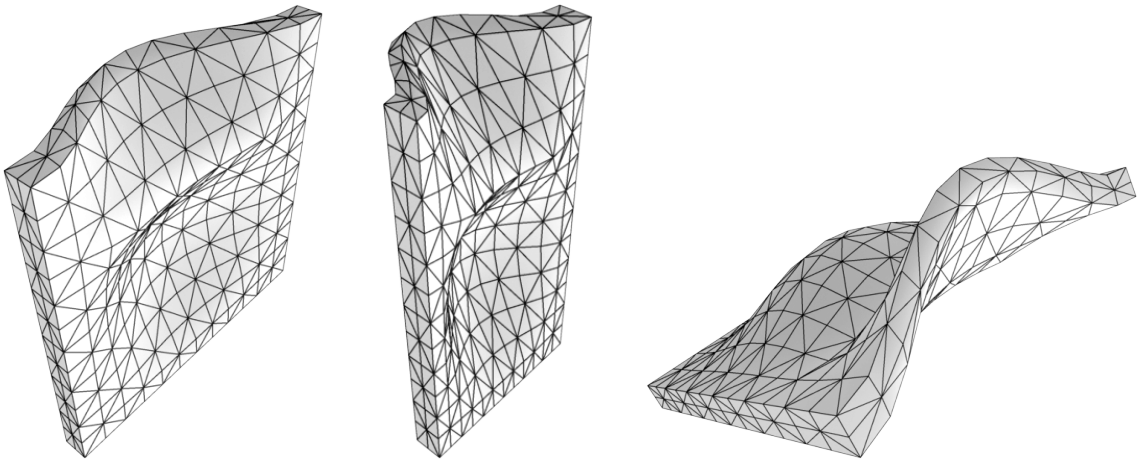


Figura 10.3. El objeto A muestra la malla original sin modificación. El objeto B muestra la malla después de aplicarsele una modificación relativa en la escala de uno de los ejes cartesianos. El objeto C muestra la malla después de un giro.

Esto se consigue mediante la modificación del transform, nodo padre del objeto. De esta forma, como este nodo dirige los parámetros de la figura final, al modificarlo también se aplicará sobre las deformaciones y otros efectos que sobre ella hay aplicadas.

La figura 10.4. muestra esquemáticamente la distribución de los nodos en el DG y en que punto se produce la variación de sus atributos:

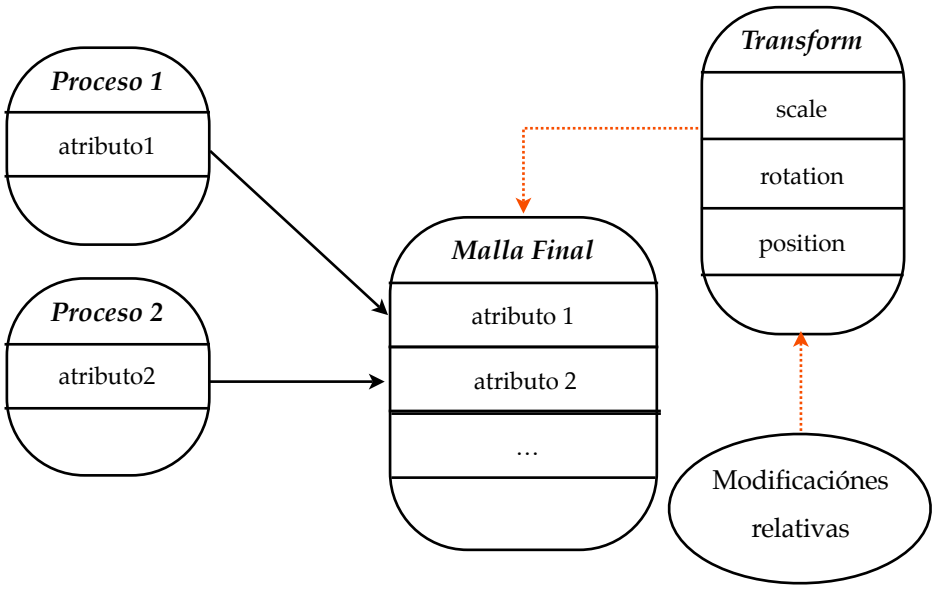


Figura 10.4. Diagrama explicativo de la relación entre el nodo transform y la malla de estudio (shape).

Como se ve en la figura, la malla de estudio recibe los procesos de modificación sobre sus atributos, resultando la malla definitiva al final del DG. Esta malla final (nodo shape) viene dirigida por un nodo DAG padre (nodo transform). Modificando este nodo padre, por tanto, se modifica en la misma medida cada uno de los atributos de la malla final sobre los cuales se aplicaron los procesos. De esta forma, la rotación por ejemplo, se realiza de forma relativa al espacio objeto y no de forma absoluta en el espacio general.

5. Definición de los Materiales

Los materiales que componen una escena son elementos claves que determinarán la apariencia de la imagen renderizada final. Cuando la finalidad de estos materiales es analítica y explicativa, como en el caso de Elesys, este aspecto toma aún mayor relevancia y deja de tener un carácter puramente estético.

Elesys hace uso de las enormes posibilidades de sombreadores y materiales de *Mental Ray* para elevar a un nivel superior los resultados obtenidos. A pesar de la finalidad artística con la que en principio se concibió *Mental Ray*, Elesys amplía sus usos al campo analítico buscando nuevas posibilidades.

Material Background

Este material se concibió para los mallados secundarios que actúan como fondos contextualizadores de la escena. Como se comentó en el punto anterior al hablar de estos elementos, al tener finalidad secundaria, no puede contar con colores llamativos que distraigan al usuario del motivo principal.

De esta forma se ha optado por un color gris oscuro. Su tonalidad neutra permite que haya un gran contraste con la malla de estudio. El color se consigue mediante un efecto *amb occlusion* que, si bien aumenta en cierta medida el tiempo de render, da al escenario un aspecto más fotorealista. La proyección de sombras

que permite el material, ayuda a situar en el espacio 3D la figura desde el punto de vista del espectador.

Consta de un sombreado *lambert* que le da aspecto mate, similar al plástico mate, consiguiendo así ausencia de reflejos y efectos imprevistos no deseados.

Material Color de Relleno

Este material se diseña para permitir al usuario colorear la malla principal con colores sólidos.

Elesys da completa libertad, en este sentido, al usuario que podrá elegir cualquier tono de la paleta de colores RGB/HSV a través de la interfaz de usuario (como se explicará en capítulos posteriores).

Al igual que el material background, hace uso de un sombreador *lambert* para evitar reflejos poco útiles en esta situación.

Materiales Maxwell Render

El motor de render principal usado por Elesys, como comentamos anteriormente, es *Mental Ray*. Opcionalmente, la aplicación da la opción de realizar render fotorealistas usando el motor *Maxwell Render*. Al ser un software comercial, Elesys no obliga a poseer este motor, sino que da la posibilidad de utilizarlo en el caso de tener comprada la licencia.

Maxwell Render es un motor de render basado en las leyes físicas de la luz real. Los algoritmos y ecuaciones reproducen el comportamiento de la luz de una manera completamente exacta. Todos los elementos de Maxwell, como los sistemas de iluminación, materiales físicos y cámaras, están enteramente basados en modelos físicamente exactos.

El método de cálculo de Maxwell converge siempre a la solución correcta sin introducir artefactos debido al hecho de que es un rénder imparcial. Puede capturar completamente todas las interacciones ligeras entre todos los elementos en una escena. Se realizan todos los cálculos de la iluminación usando la información espectral y datos de la alta gama dinámica.

Maxwell utiliza materiales con la extensión mxm. Estos materiales responden a un comportamiento real, exacto al mundo físico. Dentro de Maxwell, los materiales son modelados de una forma físicamente correcta. Son definidos a través de sus curvas BSDF (*Bidirectional Scattering Distribution Function* o Función de Distribución de Dispersión Bidireccional).

Los materiales también tienen la ventaja de tener todas las propiedades dentro de un solo fichero lo que permite que un mismo fichero mxm trabaje en múltiples plataformas. Esto hace que el mismo material funcione exactamente igual en cualquier sistema operativo en el que se ejecute Elesys.

Así, se ha creado una serie de materiales predefinidos. Estos van desde materiales de fluidos, hasta plásticos, hormigón, cristal, etc.

Una vez el usuario seleccione un material para el mallado principal y ordene renderizar, Elesys enviará la información necesaria y lanzará automáticamente el motor de Maxwell Render.

Contorneado

Con el fin de no perder la referencia del estado de la malla, desde el punto de vista de sus nodos y elementos, Elesys ofrece la posibilidad de visualizar la figura contorneada mediante el renderizado de contorno.

La figura inferior muestra la diferencia entre un render con y sin contorneado.

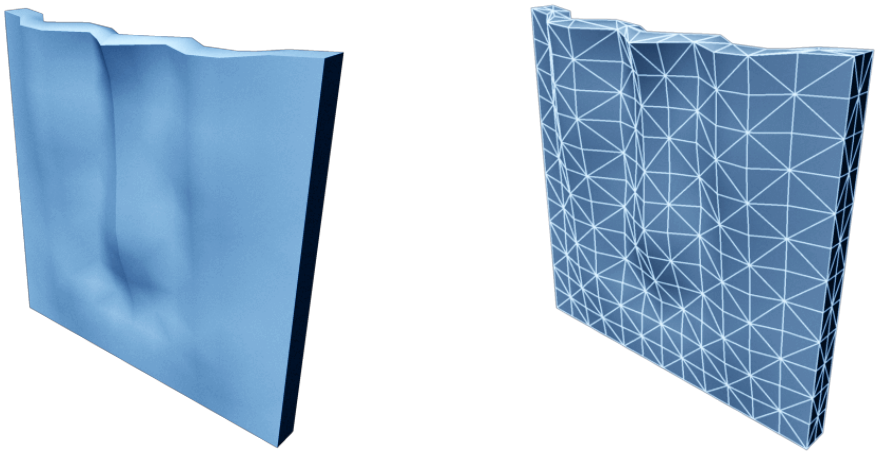


Figura 10.5. Mallado sin contorneado a la izquierda y con contorneado a la derecha

El contorneado aporta facilidad en el análisis al poder localizar visualmente en todo momento los nodos y elementos de la malla. Igualmente facilita la comprensión del movimiento al verse la deformación de cada elemento.

Existen una serie de parámetros de contorneado que se pueden modificar a través de la interfaz de usuario. Así puede cambiarse el grosor, la opacidad y el color.

Hay que destacar que el contorneado estará disponible para todos los tipos disponibles de render en Elesys excepto para render fotorealistas con Maxwell Render. El propio objetivo del uso de Maxwell Render, es decir, el representar una imagen o un video para una simulación que se acerque lo más posible a parámetros físicos reales, hace poco coherente el uso de contorneado. Éste sólo conseguiría desvirtuar las imágenes haciendo que sea evidente su artificialidad.

Material Color Por Vértices

La creación de un coloreado por vértices es el mayor avance de Elesys desde el punto de vista de la representación. Proporciona una herramienta, ya disponible en otras aplicaciones, pero con un enfoque mucho más eficiente y dinámico.

El coloreado por vértice permite un coloreado de la malla en función de un determinado parámetro asociado a cada nodo. Usando una escala de color que responda a la magnitud de dicho parámetro, el usuario puede identificar los puntos críticos, puntos nulos, etc.

En la figura 10.6. se muestra una malla con color por vértices asignada. En ella se observa como los puntos de color rojo muestran un valor absoluto mayor variando a naranja, amarillo, celeste y azul a medida que desciende el valor.

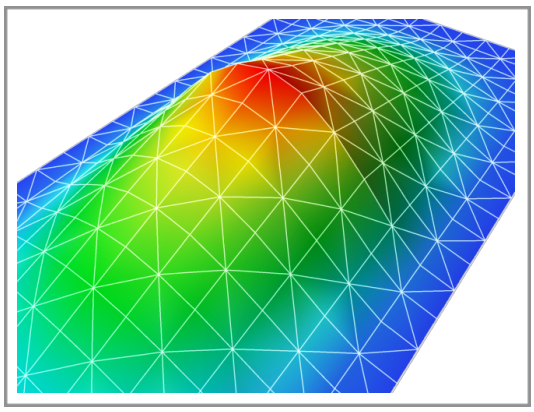


Figura 10.6. Mallado con color por vértices asignado.

Los detalles técnicos de la implementación de este sistema será comentado en los puntos siguientes de este mismo capítulo. Sin embargo, desde el punto de vista de los materiales, el color por vértices se ha implementado mediante la creación de un nodo *Color Per Vertex* de Mental Ray. Desde el punto de vista del DG, el ColorSet del mallado alimentará el nodo *Color Per Vertex* proporcionando a este el color de cada vértice en función del valor de la variable estudiada. Finalmente el nodo *Color Per Vertex* dará la información al nodo de sombreado. La figura 10.7. esquematiza este proceso.

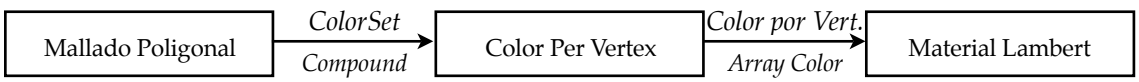


Figura 10.7. Visión esquemática de transvase de información entre nodos del DG

La representación en tiempo real del sombreado por vértices en figuras en movimiento representan uno de los grandes logros de Elesys. Por ello a continuación se comentan sus detalles técnicos.

6. Aspectos teóricos del Color Por Vértices.

Anteriormente se comentó los detalles de las mallas poligonales explicando sus elementos constitutivos. Para este método de sombreado, como su nombre indica, toma gran importancia los vértices de la malla y los vértices de caras.

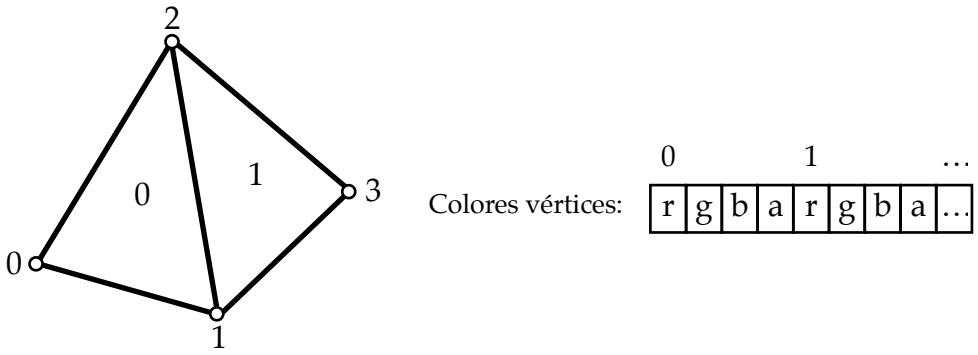


Figura 10.8. Color por vértice

Es común asociar datos a los vértices de una malla. Cada vértice tiene asociado un color, una normal, unas coordenadas uv, etc. En la figura 10.8., se asocia un color con cada vértice individual. Para simplificar la explicación, se muestra una cadena de colores. Maya no almacena los colores de esta forma pero ayuda a entender conceptualmente cómo los datos pueden ser asociados a los vértices.

Para recuperar el color asociado a un vértice, la cadena **color** es indexada usando los índices de los vértices. Así, el color del segundo es recuperado como sigue:

```
vertex_colors [1]
```

Una cadena similar podría crearse para asociar normales, coordenadas uv u otros datos similares con los vértices de la malla. Si la malla es renderizada usando colores de los vértices ambas caras, 0 y 1, podría compartir el color de los vértices

1 y 2. Cambiando el color del vértice 2, se puede cambiar el color de las dos caras ya que éstas comparten dicho vértice.

Así, se observa como cada cara comparte los colores que tienen los vértices de mallas que las forman. Sin embargo, si se quiere renderizar cada cara con colores diferentes, con la estructura actual no sería posible. Por ejemplo, no habría forma de renderizar la cara 0 en rojo y la cara 1 en verde. Se necesita una manera de asociar un color a las caras.

Esto sería posible si se hace distinción entre los datos asociados a un vértice de malla dado y los datos asociados a una cara. Esta distinción se crea usando **vértices de cara**.

La misma malla de la figura 10.9. se presenta ahora con los colores de vértices pero en formato vértices de cara. Es importante destacar que aunque la cara aparece en la figura como caras distintas y separadas, todavía se encuentran asociadas como antes. Por tanto, esta separación es a nivel conceptual aunque físicamente no lo estén.

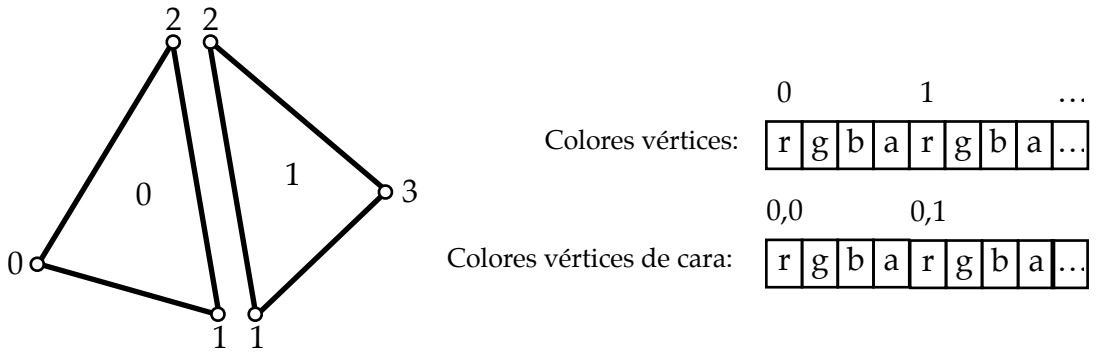


Figura 10.9. Colores de vértices de cara

Los colores de vértices de malla están aún disponibles, y por tanto, es todavía posible asociarles un solo color de tal forma que, todas las caras de alrededor de los vértice, usan este color para colorearse. Con la cadena de color de vértices de cara, ahora es posible asociar un color a un vértice o a una cara.

Volvamos al ejemplo. Como antes, hay dos caras, 0 y 1. Cada cara tiene su propia secuencia de vértices. Los índices dentro de estas cadenas de vértices son denominadas *IDs de vértices de cara*. Éstas se diferencian de los identificadores de los vértices de malla de los que tanto se ha escrito en esta memoria.

Los identificadores de vértices de la malla, tienen un rango entre 0 y el número de vértices de la malla - 1. Por su parte, el rango de los identificadores de los vértices de cara depende según en la cara que sea aplicada. Su rango es de 0 al número de vértices en la cara - 1.

Para recuperar el color del tercer vértice de la malla, ID del vértice número 2, podría usarse:

```
vertex_colors [2]
```

Para recuperar el color del segundo vértice (índice 1) en la segunda cara (índice 1), primero se debe convertir dentro de una malla relativa al ID del vértice y al par de caras. El identificador de vértice relativo a la cara (1) es mapeado en el ID del vértice de malla (3). El índice de cara permanece igual.

```
face_vertex_colors [ (3,1) ]
```

Como tal, cualquier dato asociado con el vértice de cara es indicado usando ambos identificadores, el del vértice relativo a la malla y el identificador relativo a la cara.

7. Aspectos de Desarrollo del Color por Vértice

El desarrollo de la función de computación del nodo `colorPerVtx` ha supuesto uno de los mayores retos en la creación de Elesys. En primer lugar, por tratarse de una tecnología de Mental Ray que nunca se ha usado con estos fines. Hasta la actualidad, el coloreado por vértices ha sido un método de renderizado que permitía la gestión de ese modelo de forma que se minimizara el uso de recursos

de computación. Uno de las utilidades prácticas más conocidas es la industria de los videojuegos. Mediante un render por color de vértices, por ejemplo, se puede gestionar el movimiento de personajes con un uso mínimo de GPU y CPU.

Sin embargo, en estos casos, el color por vértices era algo estático. Los vértices permanecen con los mismos datos de colores durante gran parte o toda la representación del modelo. Elesys, por el contrario, da un giro a este enfoque y, no utiliza este sistema como medio para ahorro de recursos sino todo lo contrario, lo trata como un proceso altamente dinámico. Se obliga a la máquina a variar el color de millones de vértices por segundo teniéndose que ver los resultados por pantalla en tiempo real.

Por tanto, el lector puede imaginar que se tuvo que plantear la estrategia desde cero y rediseñar un nuevo enfoque desde cero.

Atributos y gestión de datos.

El tipo de datos que va a importar el nodo `colorPerVtx` es exactamente el mismo que el normalizado para ficheros de movimiento[4]. La lectura de estos datos se realizan mediante una matriz que recolectará la información para su posterior uso por la función de computación.

Esta lectura, evidentemente, se realiza con los mismos principios de eficacia que se implantaron en la lectura del fichero malla o el fichero de movimiento. Igualmente, se hará uso de expresiones regulares y demás sistemas de gestión de errores ya usados en el resto de desarrollo de Elesys.

Función de computación del nodo.

La función de computación se puede dividir en tres etapas diferenciadas:

---

4 Puede consultarse este formato en el capítulo 8, concerniente a la verificación de ficheros, de la presente memoria. Recuerde que estos ficheros pueden hacer mención al desplazamiento o cualquier otra variable vectorial o escalar.

- Fase de cálculo de los valores de la variable en cada uno de los vértices.
- Transpolación de los valores al código de colores.
- Modificación de los vértices del mallado.

La primera fase, pasa por calcular nodo a nodo su valor por medio de la fórmula ya conocida:

$$X = VI_X + Mod_X \cdot \cos(\omega \cdot t)$$

$$Y = VI_Y + Mod_Y \cdot \cos(\omega \cdot t)$$

$$Z = VI_Z + Mod_Z \cdot \cos(\omega \cdot t)$$

$$VF = \sqrt{X^2 + Y'^2 + Z^2}$$

siendo:

VI_i = valor inicial de la variable en el punto en el eje i.

Mod_i = módulo en el eje i.

ω = frecuencia del fichero.

VF = valor final en el punto

En el caso que sea una variable escalar, sería:

$$VF = VI + Mod \cdot \cos(\omega \cdot t)$$

La segunda fase consiste en utilizar una escala de medida para transformar los valores finales de los vértices en colores. El inicio de la escala está claro, es el cero, puesto que estamos con magnitudes en valor absoluto.

El segundo dato necesario es el máximo que alcanza la variable. Elesys da dos opciones al usuario:

1. **Seleccionar un valor numérico como máximo.** Esto resulta especialmente interesante cuando exista valores de la variable estudiada que el modelo no pueda sobrepasar, como por ejemplo, el límite de rotura o el valor de presión máximo admisible. Este valor se introducirá fácilmente a través de la interfaz de usuario.

2. **Dejar que sea la propia aplicación la que seleccione automáticamente el máximo**. Este valor será el máximo que alcance cualquier vértice de la malla en el momento más favorable.

Para este segundo caso destacar que este paso, realmente, se ha realizado en paralelo con obtención del valor final VF. El sentido de realizar la obtención de VF y el valor máximo V_{max} está en tener que realizar la lectura del fichero una sola vez, de forma que se realice lo más rápido posible consiguiendo la mayor velocidad de proceso posible.

El valor del máximo se calcula como:

$$X = VI_X + Mod_X \cdot 1$$

$$Y = VI_Y + Mod_Y \cdot 1$$

$$Z = VI_Z + Mod_Z \cdot 1$$

$$VF_{max} = \sqrt{X^2 + Y'^2 + Z^2}$$

Por tanto, el valor máximo V_{max} será el mayor de los VF_{max}.

Una vez se tiene el cero y el valor máximo, sólo queda distribuir la escala de colores.

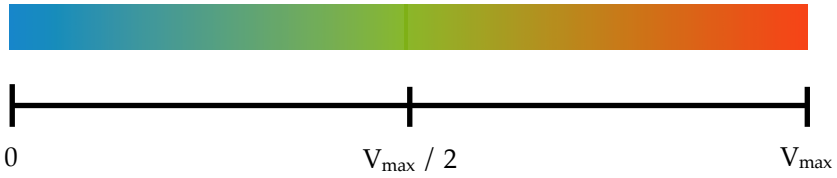


Figura 10.10. Escala de colores

Para confeccionar esta escala de colores, se ha elegido la gama más usada por la mayoría de los programas de postproceso del mercado. Como se ve en la figura 10.10., consiste en tonos azules para los valores más bajos, verdes para los intermedios y naranja-rojo para los valores más alto.

A la hora de escribir el código se ha utilizado la clase **MColor** que proporciona Maya. El tipo de paleta elegido ha sido el HSV puesto que permite una variación más cómoda entre estos valores de colores que su comparación con el sistema RGB.

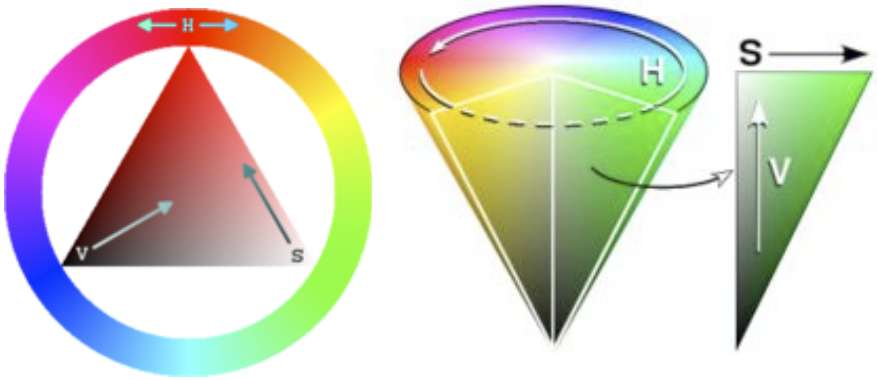


Figura 10.11. Variaciones de la paleta HSV variando sus tres parámetros

---

El modelo HSV fue creado en 1978 por Alvy Ray Smith. Se trata de una transformación no lineal del espacio de color RGB y se puede usar en progresiones de color como la que nos ocupa en este caso.

El modelo HSV (del inglés *Hue*, *Saturation*, *Value* – Tonalidad, Saturación, Valor), también llamado HSB (*Hue*, *Saturation*, *Brightness* – Tonalidad, Saturación, Brillo), define un modelo de color en términos de sus componentes constituyentes en coordenadas cilíndricas:

- Tonalidad, el tipo de color (como rojo, azul o amarillo). Se representa como un grado de ángulo cuyos valores posibles van de 0 a 360° (aunque para algunas aplicaciones se normalizan del 0 al 100%). Cada valor corresponde a un color. Ejemplos: 0 es rojo, 60 es amarillo y 120 es verde.

- Saturación. Se representa como la distancia al eje de brillo negro-blanco. Los valores posibles van del 0 al 100%. A este parámetro también se le suele llamar "pureza" por la analogía con la pureza de excitación y la pureza colorimétrica de la colorimetría. Cuanto menor sea la saturación de un color, mayor tonalidad grisácea habrá y más decolorado estará. Por eso es útil definir la insaturación como la inversa cualitativa de la saturación.

- Valor del color, el brillo del color. Representa la altura en el eje blanco-negro. Los valores posibles van del 0 al 100%. 0 siempre es negro. Dependiendo de la saturación, 100 podría ser blanco o un color más o menos saturado.

Puesto que un color de la paleta HSV se selecciona en Maya mediante tres flotantes (tonalidad, saturacion y valor de color), se ha comprobado que basta ligeras variaciones de las dos últimas según en la región de la escala en la que se encuentre, siendo la tonalidad la variable realmente importante.

Así, hay que reajustar la escala (0 - V_{max}) a la escala de la parte de la variación de la tonalidad del HSV (0 - 240). La escala completa de tonalidad del HSV va del 0 al 360. Sin embargo sólo nos interesa la parte que va desde el color azul al color rojo (0 - 240). Este reajuste se puede ver gráficamente en la figura 10.12.

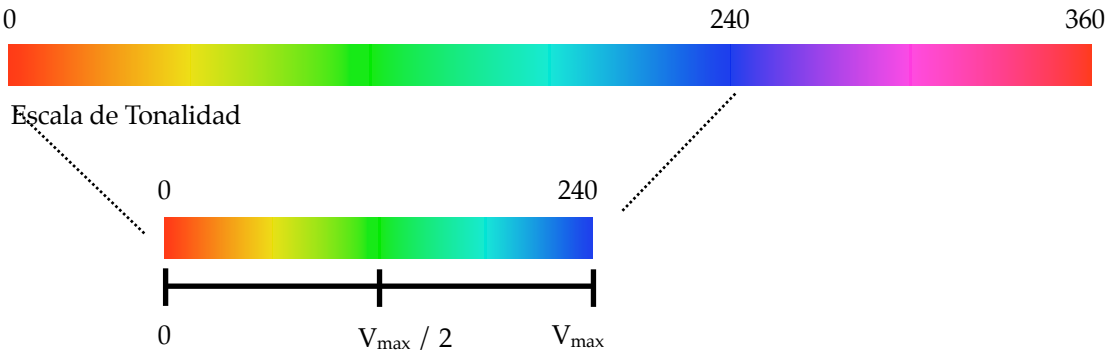


Figura 10.12. Proceso de reajuste de escalas

Hay que notar el detalle de que los valores mayores (240) corresponde al azul que tiene que representar los valores menores en los vértices. Igualmente, el valor menor 0, corresponde al rojo que es el valor mayor en el vértice V_{max}. Por ello debe realizarse un giro, que se consigue muy fácilmente con la fórmula:

$$H = 240 - V_E$$

Siendo V_E el valor de la variable en el nodo después de realizarse el escalado.

Para terminar, sólo queda aplicar este color a la malla. Para este fin se ha utilizado la clase **MFnMesh** de Maya. Mediante un algoritmo, en el que se ha buscado la mayor rapidez, se consigue seleccionar el vértice con el que trabajar y modificar su color con la función `setVertexColors`.

La velocidad conseguida en la implementación de este proceso de modificación de colores posibilita que pueda verse en el visor, en tiempo real, la variación del sombreado por los colores correspondientes incluso para modelos con gran número de vértices.

8. Parámetros de Render

En cualquier motor de render, el número de parámetros, opciones, filtros y otras herramientas a utilizar es enorme. La necesidad de tomar esta gran cantidad de decisiones acerca de los parámetros de un render y la enorme cantidad de

complejos términos que se manejan puede desanimar al usuario novel en el mundo de la infografía.

Puesto que Elesys es una aplicación destinada al campo de la ingeniería, es de esperar que el usuario medio no tenga este tipo de conocimientos por lo que no tendrá la posibilidad de tomar estas decisiones. Si a esto sumamos la filosofía de simplicidad que se le pretende dar a toda la aplicación deja patente la necesidad de evitar al usuario la configuración de cada representación.

Por ello se ha establecido una serie de configuraciones a medida que simplificarán todo el proceso. Es necesario establecer un compromiso óptimo entre calidad y tiempo de render.

Tomando varias escenas tipo, con un grado de complejidad variable y necesidades de representación varias, se realizaron estudios de rendimiento y tiempo de cálculo. A través de este estudio y analizando la calidad de resultados se establecieron los parámetros más adecuado para los render en Elesys, tanto en Mental Ray como en Maxwell Render.

La conclusión de este estudio para el motor Mental Ray puede resumirse en los siguientes puntos:

- El uso de un primer proceso de Rasterizer, que permite una rápida representación con poca pérdida de calidad.
- Un segundo efecto mediante el algoritmo de *Raytracing* que permite aumentar la calidad del render mediante el cálculo de efectos globales de iluminación deseados como pueden ser reflexiones, refracciones o sombras arrojadas.

  Sin embargo, los parámetros para este proceso no pueden ser muy altos pues, llegados a ciertos niveles, producen un aumento de tiempo de render muy alto.

- Cálculo de sombras causadas por la iluminación de la escena. Se ha optado por un método simple de cálculo al ser el más equilibrado en calidad/ tiempo de proceso.

- Se ha desechado *Motion Blur* puesto que aumentaría la duración del render y restaría nitidez a la representación.

- Igualmente se ha eliminado algoritmos como *Caustics* o *Final Gather* que, si bien tiene efectos muy positivos en la calidad de la imagen final, requieren de tiempo inadmisible para una representación equilibrada.

- El uso de los contorneadores serán indicados por el usuario puesto que no aumentan o disminuyen el tiempo de render y se entiende que es una variable cuya validez y configuración depende del caso concreto de estudio. Para su selección se hará uso de la interfaz de usuario.

.

Interfaz Gráfica

La interfaz de usuario es la manera en la que la persona que utiliza una computadora se comunica con ella. La modalidad más conocida es la interfaz gráfica de usuario. Esta permite una interacción amigable entre la persona y el ordenador.

La interfaz gráfica de usuario (en inglés *Graphical User Interface*, GUI) es un tipo de interfaz de usuario que utiliza un conjunto de imágenes y objetos gráficos para representar la información y acciones disponibles. Habitualmente las acciones se realizan mediante manipulación directa para facilitar la interacción del usuario con la computadora.

La GUI es una evolución natural de la línea de comandos de los primeros sistemas y es pieza fundamental en un entorno gráfico. La historia reciente de la informática está indisolublemente unida a las interfaces gráficas, puesto que los sistemas operativos gráficos han ocasionado grandes consecuencias en la industria del software y del hardware.

La necesidad de dotar a Elesys de una interfaz gráfica surge del objetivo de hacer la aplicación más accesible para el uso de los usuarios comunes. Con ella se evitará que los futuros usuarios puedan acceder al sistema, sin tener que pasar por el tortuoso proceso de aprendizaje para manejar un entorno bajo línea de comandos.

1. Interfaz MEL

MEL puede usarse para crear una gran variedad de GUI's. De hecho, la interfaz completa de Maya está construida y gestionada usando comandos MEL. Estos mismos comandos están disponibles para los desarrolladores, para diseñar y construir interfaz de usuarios a medida.

En el caso que nos ocupa, era necesario proporcionar una interfaz de usuario a la funcionalidades programadas en Elesys. En un típico entorno de desarrollo, la creación de interfaces de usuarios puede ser una larga y tediosa tarea. Aunque Maya no proporciona una fácil visualización de la creación y disposición de los elementos de la interfaz, sí proporciona un rico conjunto de comandos de creación y gestión de interfaces.

La mayor ventaja de MEL es su habilidad de ejecutar comandos inmediatamente, con lo que se puede hacer rápidamente prototipos de interfaces de usuario. Una vez que el script que genera la interfaz está preparado, puede ser ejecutado, y la interfaz se mostrará inmediatamente. Es posible volverse posteriormente al script y editar las capas y los controles de la interfaz como se necesite.

Una gran ventaja de usar MEL para todas las interfaz de Elesys, es que permite crearlas sin preocupaciones acerca de su funcionamiento, a través de los diversos sistemas de interfaz de los distintos sistemas operativos en los que se desarrolla la aplicación. Cada sistema operativo, normalmente, tiene su propio sistema de ventanas con su propio API y particularidades. Usando MEL para crear los

elementos de la interfaz, libera al desarrollador de escribir código de interfaz para cada una de las plataformas. Tenemos también la garantía de que, una vez que se ha escrito la interfaz en MEL, ésta se mostrará y se comportará similarmente en cada plataforma.

Los comandos MEL que permiten la construcción de la interfaz de usuario son parte del *Extended Layer Framework* (*ELF*). MEL incluye un rico conjunto de elementos de interfaz incluyendo ventanas, botones, deslizadores, etc.

2. Conceptos Fundamentales

Para comprender cómo se crean interfaces, se cubrirá algunos de los conceptos básicos a través de los comandos más comunes.

El comando `window` permite la creación de una ventana donde posteriormente se incorporan los controles y demás elementos necesarios. Si no se especifica ningún valor para sus parámetros de creación, Maya usará los valores por defecto. Por ejemplo, al escribir:

```
window;
```

Maya crea una ventana denominada **window1**. Una ventana, por defecto, que no es visible, por lo que se necesita hacerla visible explícitamente. Para ello se ejecuta lo siguiente:

```
window -edit -visible true window1;
```

La ventana ahora sí se muestra. Se puede editar las propiedades de un elemento de la interfaz mediante el uso de la etiqueta `-edit`. En este caso, el parámetro `visible` es encendido, haciendo la ventana visible. La ventana cuenta con numerosos parámetros, incluyendo su posición, tamaño, si se redimensiona automáticamente, etc.

Destacar que, cuando se llama al comando `window`, el nombre de la ventana, **window1** es especificado. A todos los elementos de la interfaz de usuario se le asigna un nombre cuando son creados. Si no se le asigna un nombre explícitamente, por defecto se utiliza uno por defecto. Cuando se modifica o se referencia un elemento de la interfaz, se debe incluir su nombre para que Maya conozca exactamente a qué elemento se está refiriendo el código. Si se crea un nuevo elemento de la interfaz con el mismo nombre que posee un elemento ya existente, Maya mostrará un error.

Veamos el siguiente ejemplo:

```
window -widthHeight 200 300 myWindow;
columnLayout myLayout;
button -label “First” myButton;
showWindow myWindow;
```

Figura 13.1. Ventana creada con el código adjunto.

La ventana **myWindow** se muestra por pantalla. Contiene un solo botón etiquetado como **First**. Destacar, cómo a cada uno de los elementos de la interfaz se le asignó un nombre cuando fueron creados. El `window` es llamada **myWindow**, la `columnLayout` es denominado **myLayout**, y el botón es denominado **myButton**. Dados nombres exactos, ahora pueden referirse a ellos de forma precisa.

Antes de que un elemento sea añadido a la ventana, Maya debe conocer cómo debe colocarlos dentro de ella. Esto se hace mediante la colocación de elementos *layout* a la ventana. El *layout* (control de disposición) especifica cómo se coloca todo elemento subsecuente. Hay muchas formas diferentes de colocar los elementos. En este caso, se ha añadido un disposición `columnLayout` a la ventana

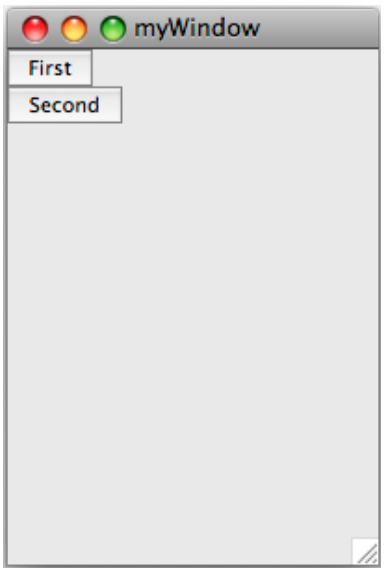


Figura 13.2. Ventana con un segundo botón añadido.

del ejemplo de la figura 11.1. Esta disposición coloca los elementos en una sola columna. Para ver esto, se puede añadir otro botón a la `columnLayout`.

```
button -label "Second" -parent myWindow|
myLayout;
```

Un botón etiquetado como **Second** se añade a la ventana. Puesto que ha sido añadido al `columnLayout`, es automáticamente colocado bajo el primer botón.

Cuando se añaden elementos de interfaz a la ventana, se crea una jerarquía. La cima de la jerarquía es la propia ventana **myWindow**. Bajo ésta encontramos la disposición **myLayout**. Después, estarán los dos botones. Cuando un elemento está bajo otro, es considerado el hijo del primero. Los dos botones son hijos de elemento **myLayout**. Pueden crearse jerarquía arbitrarias de elementos mediante la suma de hijos.

Una vez asumido que se tiene una jerarquía de elementos de interfaz, se necesita especificar la ruta completa para referirnos a uno específico. La ruta completa para la disposición es **myWindow|myLayout**. Destacar que, ésta es la misma notación usada para especificar la ruta completa a un objeto de la escena (ruta DAG), sin embargo, que no hay espacio entre los nombres de los elementos.

Las ventanas están siempre en la cima de la jerarquía. Como son elementos de la escena, es posible tener dos elementos con el mismo nombre, mientras sus rutas difieran. Por ejemplo, es válido tener dos disposiciones con el mismo nombre local **myLayout**, mientras existan en distintas jerarquías. Por ejemplo **win1|myLayout** y **win2|myLayout** son dos disposiciones distintas y válidas.

Cuando se añade un nuevo elemento, se debe siempre especificar el padre. Sin embargo, esto requiere más trabajo por parte del programador, que debe mantener en su cabeza el nombre de los elementos individuales. Para simplificar esta tarea de añadir elementos, Maya introduce el concepto de los *padres por defectos* (*default parent*). Cuando no se especifica padre, se utiliza el padre por defecto. En pasos previos, se ha creado una ventana, **myWindow**, después se ha creado un `columnLayout`. No se han especificado padre para el `columnLayout`, aún así se le emparentó automáticamente con la ventana.

Cuando un elemento de la interfaz se crea, automáticamente se genera el padre por defecto para los elementos este tipo. Existe un padre individual por defecto para ventanas, disposiciones, menús, etc. Un padre por defecto existe sólo para elementos que puede tener otros elementos como hijo. Puesto que una ventana puede tener disposiciones, menús, etc. como hijos, cuando son creados automáticamente se genera la ventana padre por defecto. Cuando se crea una disposición, se convierte en el hijo de un padre por defecto, una ventana. La nueva disposición se toma como padre por defecto para aquellos elementos que utilicen a las disposiciones como padres. Por tanto, hay una amplia variedad de padres por defecto dependiendo del tipo de elemento que es creado.

3. Elementos MEL de la interfaz de usuario

A continuación, se expondrá los elementos MEL más utilizados para la creación de interfaces de usuario.

Ventanas.

Una ventana es el elemento de interfaz básico y es el más usado para contener un amplia variedad de otros elementos. Una ventana puede contener una barra de título, así como botones de maximizar y minimizar. Una ventana también puede ser posicionada y redimensionada. Por defecto, una ventana es invisible. La

razón de esto, es que se puede crear una ventana vacía y mostrarla, después comenzar a añadir elementos, calcular la disposición, etc. para cada uno de los elementos y después mostrar los elementos de forma interactiva. Cuando se crea una ventana invisible y después se añade elementos, se muestra mucho más rápida, puesto que no tienen que mostrar interactivamente cada elemento a medida que es añadido.

Disposiciones

Hay una amplia variedad de disposiciones. Cada una, responde a un propósito común: posicionar o dimensionar los elementos de la interfaz añadidos a ellos. Se pueden crear esquemas complejos de disposiciones, jerarquizando diversos tipos de éstas.

Cada ventana debe tener al menos una disposición. Esta primera disposición toma toda el área del cliente de la ventana. Ésta es el área de la ventana que no incluye el título, la barra de menú, bordes, etc. La excepción a esto es el `menuLayout`, que lo extiende verticalmente a la altura de la barra de menú. Con ésta no se asumirá el control del área entera del cliente.

Alguna de las disposiciones más usadas son expuesta a continuación:

- **Disposición en columna (*columnLayout*):** la disposición en columna coloca todos los hijos en una sola columna, uno detrás del otro. Es posibe definir el ancho de la columna y el alto de la fila, así como con quién será fijado cada hijo.

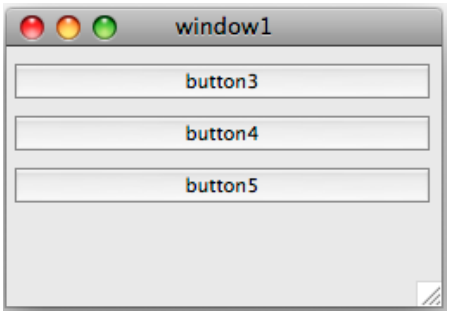


Figura 13.3. Botones colocados en disposición por columna.

- **Disposición en fila (*rowLayout*):** la disposición en fila coloca cada elemento hijo en una fila. Se debe especificar el número de columnas en la fila. Por defecto, ésta es cero, así que cuando se agrega a un hijo, se obtiene un error similar al siguiente:

```
// Error: file: -/learning.mel line 4
    Too many children in layout: rowLayout1 //
```

Para corregir esto, simplemente se ha de incrementar el número de columnas cuando el `rowLayout` es definido.

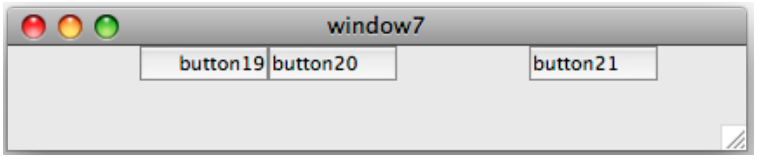


Figura 11.4. Botones colocados en disposición por fila.

Es posible determinar la alineación, el desfase y la adjustabilidad de cada columna individual.

- **Disposición en rejilla (*gridLayout*):** la disposición en rejilla posiciona todos los hijos en una serie de filas y columnas. A medida que se añaden elementos, estos se van acomodando a través de la suma de más filas y columnas según se van necesitando. Este comportamiento puede prevenirse a través de la fijación del número de filas y columnas. La anchura y altura de las celdas en la rejilla puede también ser especificada.

Figura 11.5. Botones dispuesto en disposición por rejilla.

- **Disposición en formularios (*formLayout*):** esta disposición es la que muestra el esquema más flexible, ofreciendo un gran número de opciones de alineación y fijación. También proporciona posicionamiento relativo o absoluto de los elementos hijos. Cuando un elemento se añade al `formLayout`, éste no tiene posición por defecto. Su posición debe ser especificada después. Así, el procedimiento general es añadir todos los elementos hijos al `formLayout`, y después editar sus características para finalmente posicionarlos.

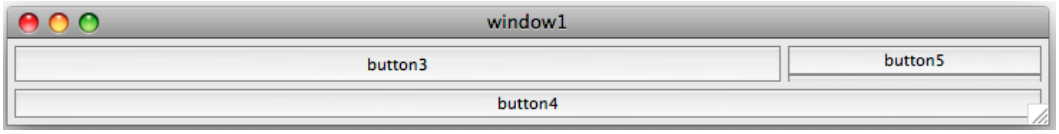


Figura 11.5. Botones colocados en disposición por formularios.

- **Disposición en marco (*frameLayout*):** cuando se necesita tener una región de la interfaz que pueda expandirse y colapsarse, se usará esta disposición. El `frameLayout` típico contiene un botón de expandir/colapsar. Colapsando se reduce el tamaño del marco y expandiendo se muestra todos los elementos que están en él. El `frameLayout` tiene opciones de etiquetas, bordes, márgenes, etc. Es importante destacar que esta disposición necesita otra disposición que ordene el contenido de sus hijos.

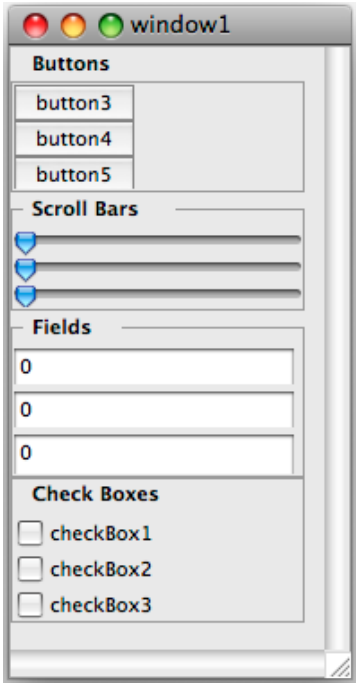


Figura 13.6. Elementos de la interfaz en disposición en marco.

- **Disposición en pestañas (*tabLayout*):** el `tabLayout` permite organizar otras disposiciones dentro de una serie de directorios. Cada directorio tienen un botón de pestaña en la parte superior que puede seleccionarse. La primera vez que la pestaña es seleccionada, se le asocia la disposición mostrada. El `tabLayout`, como el `frameLayout`, permite hacer más eficiente el uso de la misma región de la ventana para reutilizar múltiples disposiciones.

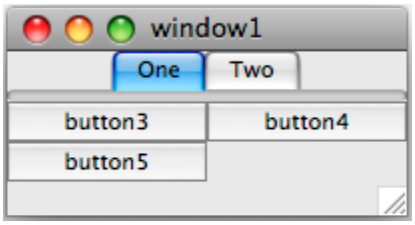


Figura 11.7. Botones en disposición por pestaña.

Similarmente a la disposición en marco, la disposición en pestaña obliga a tener otra disposición asociada a ella. Así, si se intenta añadir algún elemento sin otra disposición, habrá un error.

- **Disposición en scroll (*scrollLayout*):** cuando un elemento de la interfaz es muy grande para ser mostrado dentro de una región, puede ser usado un `scrollLayout`. Éste no proporciona en sí mismo una disposición, sino que muestra una barra de desplazamiento para las disposiciones hijas.

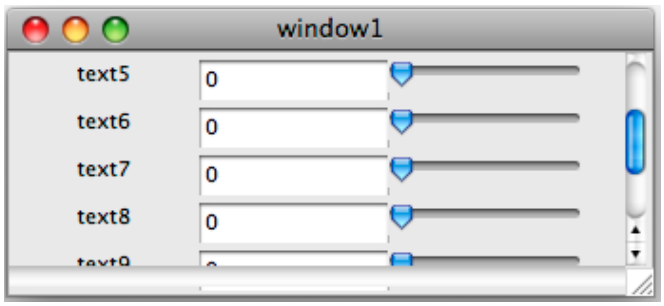


Figura 11.8. Varios elementos que pueden ser mostrados mediante el movimiento del scroll.

- **Disposición en barra de menú (*menubarLayout*):** esta disposición es diseñada para contener submenús. Posteriormente, estos submenús contiene sus propios `menuitems` individuales. El `menuBarLayout` puede ser usado para crear un menú general para una ventana o un menú separado de la disposición existente.

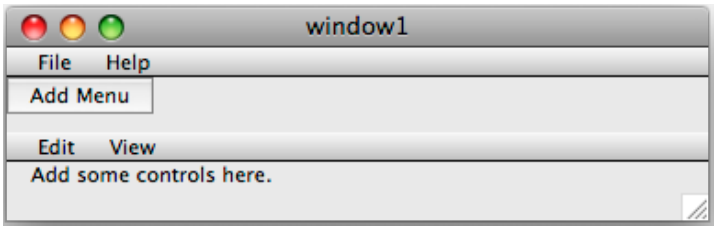


Figura 11.9. Ventana con barras de menú.

Controles.

Los controles son elementos individuales de la interfaz de usuario con los que el usuario interactúa. Los controles incluyen botones, deslizadores, campos de texto, etc. Los controles más comunes en Maya son:

- **Menús:** pueden ser añadidos a la barra de menú principal de la ventana o un `menuBarLayout`. El siguiente ejemplo muestra como crear una barra de menú principal.

- **Botones:** son controles que pueden adoptar múltiples formas, pero su propósito fundamental es proporcionar un control que el usuario pueda clickear para iniciar alguna acción. Esta acción es definida con una o más sentencias MEL.

  En cuanto a su apariencia, el botón puede aparecer con una etiqueta de texto, con una imagen o con un icono. Si se necesita un botón que muestre una imagen por etiqueta, se debe usar el comando `iconTextButton`.

- **Cajas de chequeos (*check boxes*):** es un pequeño botón, con la particularidad de que se utiliza para indicar si algo está encendido o apagado. Clickeando el *check box* se provoca que modifique su estado. Si está apagado se encenderá y viceversa.

  Además del comando `checkBox` que crea un *check box* estándar, existe también el comando `SymbolCheckbox`, que crea el *check box* con un icono. También tenemos `iconTextCheckbox`, que puede tener una combinación de la etiqueta y el icono.

- **Botones radio (*radio buttons*):** le permite elegir un elemento de un grupo. Puesto que conoce que el *radio buttons* pertenece a un grupo dado, necesita primero crear un `radioCollection`. Esta colección contendrá al *radiobutton*.

- **Texto:** para mostrar un sola línea de texto editable o de solo lectura, utilice el comando `textField`. El siguiente ejemplo crea un campo de texto editable. Es importante siempre presionar **Enter** cuando se finalice de editar el cuadro de texto. Si el usuario no presiona **Enter**, el cambio realizado en el texto será ignorado.

  Si se necesita mostrar múltiples líneas de texto, se utiliza el comando `scrollField`. Un elemento `scrollField` puede mostrarse como solo lectura o texto editable.

- **Listas :** para mostrar listas de elementos de texto desde la cual, el usuario pueda seleccionar alguno de ellos, utilice el comando `textScrollList`. Es posible configurar el `textScrollList` de forma que el usuario pueda seleccionar un solo elemento o varios de ellos.

- **Grupos:** a menudo se necesitará crear un grupo de elementos para mostrar algún dato dado. Por ejemplo, puede que se quiera mostrar un número en coma flotante, posteriormente proporcionar un deslizador y una caja

editable para manipularlo. Afortunadamente Maya proporciona algunos comandos muy convenientes que permiten crear y gestionar grupos enteros de elementos.

La siguiente tabla muestra todos los comandos que pueden crear y gestionar grupos:

Grupo	Descripción
colorIndexSliderGrp	se traslada a través de un grupo de colores
colorSliderGrp	edita un color con una muestra o deslizador
floatFieldGrp	Edita un flotante con un campo
FloatSliderGrp	Edita un flotante con un campo o deslizador
intFieldGrp	Edita un entero con un campo
intSliderGrp	Edita un entero con un campo o deslizador
radioButtonGrp	Colección de radiobuttons
textFieldGrp	Edita un texto con una etiqueta y campo

- **Imágenes:** para mostrar una imagen bitmap estática, se puede utilizar tanto el `picture` o `image`. Bajo Windows, el comando mostrará ficheros `.xmp` y `.bmp`, mientras que bajo Linux y Mac OS X mostrará sólo fichero `.xmp`. El comando `image` puede mostrar estos comandos y más.

- **Paneles:** pueden ser añadidos a una ventana usando uno de los comandos paneles: `outlinerPanel`, `hardwareRenderPanel`, `modelPanel`, `nodeOutliner`, `spreadSheetEditor` y `hyperPanel`.

4. Principios de diseño

Previamente al diseño de la interfaz, nos planteamos la filosofía de funcionamiento que ésta debe tener. Estos principios son muy útiles a la hora de entrar en la verdadera creación de la GUI. Sirven de guía, de hoja de ruta, que evitará que nos alejemos del camino correcto llevados por aspectos secundarios como la pura belleza estética de la interfaz.

La filosofía de diseño debe ser principios obtenidos mediante la abstracción de los procesos a realizar y gestionar por la GUI. Por tanto, no deben ser modificados por argumentos relacionado con la creación de la misma.

Podemos resumir estos principios como:

1. **Modularidad:** los procesos deben ser separados en módulos acorde a los resultados que se obtienen. Cada proceso individual se debe poder realizar sin salir del módulo en el que está colocado. Se persigue facilitar la localización de cada proceso y realizar una buena estructuración del sistema.

2. **Minimalismo:** la presentación de la interfaz debe ser concisa y directa. La simplicidad de la información evita distracciones del usuario a causa de elementos que no aporten utilidad o compliquen su uso.

3. **Claridad:** los elementos se deberán mostrar de forma ordenada y clara, respetando márgenes, distancias y proporciones.

4. **Minimización de Etapas:** el usuario debe llevar a cabo el proceso que tenga como objetivo de forma que el número de pasos que tenga que realizar sea mínimo.

5. **Automatización:** se debe evitar la realización por parte del usuario de aquellas tareas que pueda hacerse cargo el sistema de forma automática.

5. Descripción de la Interfaz de Elesys

La interfaz de Elesys se compone de un visor, donde visualizar el resultado de los procesos realizados, y de una ventana de operaciones donde se realizarán las modificaciones y operaciones sobre el modelo.

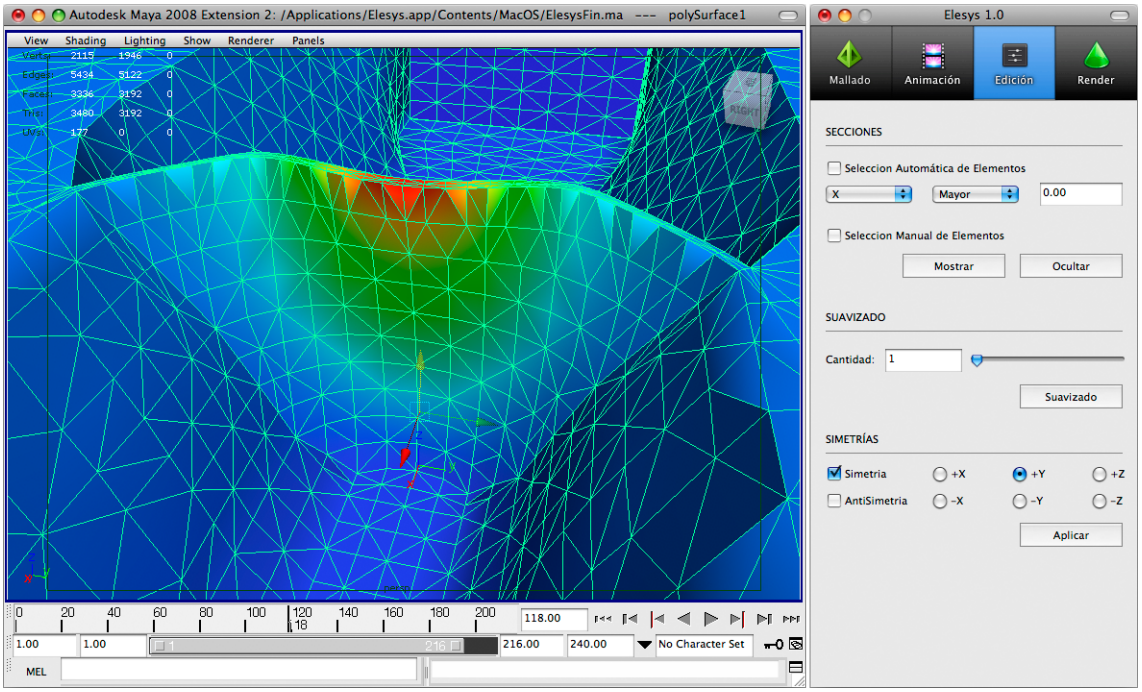


Figura 11.10. Interfaz completa de Elesys

El tamaño de ambas ventanas es la primera decisión a tomar. Para ello se estudió las resoluciones de pantallas que dispone un equipo informático de tipo medio. Tras el análisis se obtuvo la conclusión que, a día de hoy, el tamaño de pantalla media se encuentra en monitores de 19''.

Existen en el mercado dos ratios de alto por ancho de la pantalla (*aspect ratio*): formato panorámico 16:9 y formato estándar 4:3. La figura 11.11. muestra las diferencias entre ambas.

Para estos monitores, la resolución media encontrada es de 1440 x 900 pixeles para formato panorámico y 1280 x 1024 pixeles para formato estándar 4:3. Así que trabajando con estas resoluciones, se ha decidido que 340 pixeles es el tamaño

más adecuado para la ventana de operaciones. Con este tamaño, el visor queda con un tamaño mínimo de 1100 a 940 pixeles, dimensión suficiente para una buena visualización de resultados.

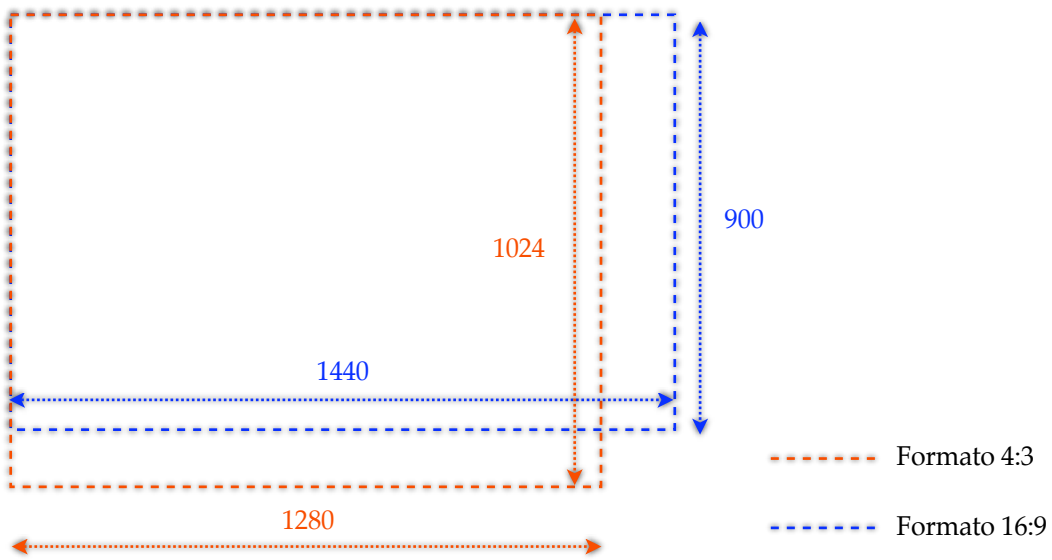


Figura 11.11. Diferencia entre ambos aspect ratios.

La ventana de operaciones, se ha diseñado cuidadosamente atendiendo a los principios de diseños expuestos. La modularidad se observa en la siguiente clasificación, en la que la ventana de operaciones se ha dividido en cuatro partes bien diferenciadas:

1. **Mallado:** se ocupa de la importación y generación del mallado, así como de la importación y eliminación de canales datos.

2. **Animación:** responsable de importar el movimiento al mallado acorde a los parámetros de tiempo, velocidad y amplitud dados en este módulo.

3. **Edición:** permite llevar a cabo operaciones sobre la malla para mejorar su visualización.

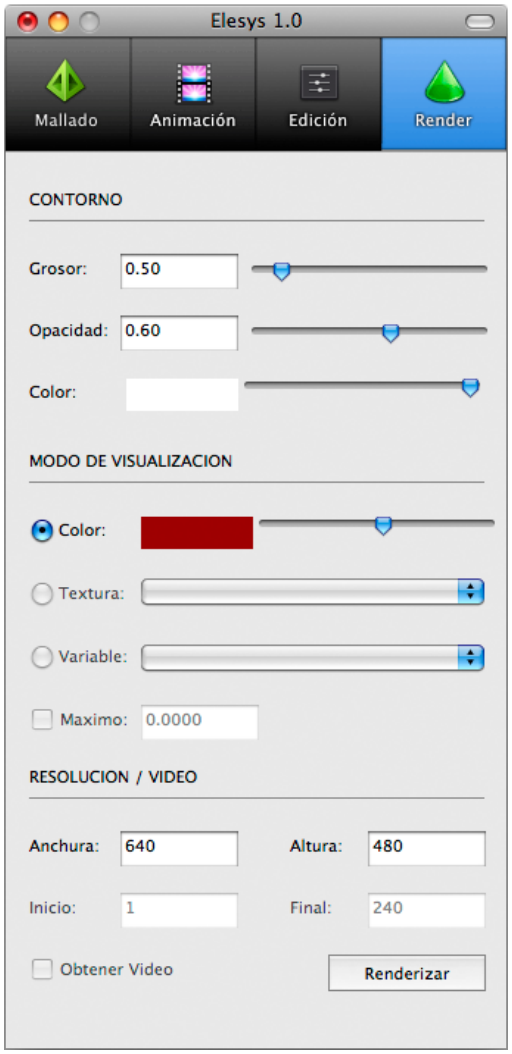
4. **Render:** se especifican los parámetros de la operación de renderizado a realizar.

Para la realización de esta separación en la interfaz se ha hecho uso de botones del tipo radio con imágenes, que colocados en la parte superior, permite ir alternando entre cada uno de los módulos. La disposición usada para estos botones es en filas, usando el `rowLayout`. Esto permite que los botones se repartan uniformemente por el ancho de la ventana.

Figura 11.12. Botones para cada módulo realizados mediante radio apoyados por iconos.

La modularidad se ha continuado dentro de cada apartado mediante el uso de títulos que separan cada proceso dentro de una misma categoría. En cada una de estas categorías se ha separado los controladores necesarios para la realización de un proceso concreto. Esta jerarquía de módulos se puede ver en la figura 11.13., en ella se muestran los títulos dentro del módulo de render.

Teniendo en cuenta la variedad de módulos a diseñar, es importante que la disposición a utilizar dentro de la ventana de operaciones sea muy flexible. Por tanto, la disposición utilizada dentro de la ventana de operaciones, bajo el `rowLayout` de los botones de módulo, es un `formLayout`. Mediante el `formLayout` se mantiene un amplio control sobre la colocación de los elementos, anclándolos a los bordes de la ventana e incluso a otros elementos de la ventana de operaciones.



Contornos: se realizan las modificaciones sobre el contorno aplicado a la figura.

Relleno: controles del color de relleno aplicado al modelo.

Resolución / Video: parámetros sobre el video o imagen a generar.

Figura 11.13. Segunda clasificación modular de los controles de la ventana de operaciones.

La definición en el script MEL de la interfaz se ha hecho de forma conjunta para todos los módulos. Es decir, todos los elementos de cada uno de los módulos existen en cada momento en la ventana. Sólo se ha tenido que especificar cuidadosamente, bajo qué condiciones, Elesys debe mostrar uno u otros.

Para esto, se ha utilizado de forma repetida la etiqueta `-visible`. Así, por ejemplo, si se hace click sobre el módulo de animación, todos los elementos relacionados con este módulo deben tener la etiqueta `-visible true`, mientras que los demás deben ocultarse mediante la etiqueta `-visible false`.

6. Control de Variables de la GUI

Merece mención aparte, los controles campo de texto (`textField`, `intField`, `floatField`, etc.). A través de ellos puede tenerse un control preciso sobre el tipo de dato introducido sin necesidad de hacer uso de otros sistemas de gestión de errores como, por ejemplo, las expresiones regulares.

Estos campos sólo admiten un solo tipo de datos de entrada. Basta con determinar el tipo de dato que se necesitará para concretar la variable relacionada con el campo, y hacer uso del apropiado comando MEL.

Otro aspecto a cuidar, en campos de números enteros y números flotantes, son los valores máximos y mínimos admitidos. Debe evitarse valores problemáticos que, aunque formalmente correctos, provoquen errores por desbordamiento o por resultados imposibles por incongruencia en relación a otros introducidos.

Pongamos como ejemplo, el campo de tiempo en segundos del módulo animación. Este campo determina la cantidad de segundos que debe calcularse de la animación de la malla. Si escribimos un valor excesivamente grande, pensemos 99.999.999 segundos, es fácil imaginar que la cantidad de información generada puede ser tan enorme que no haya espacio físico para ser almacenada por la computadora. De esta forma, es muy posible que Elesys se colapse y se cierre. Es prudente, por tanto, elegir un límite superior razonable que evite estas situaciones.

De la misma forma, pensemos en el fotograma inicial y final para una animación. Si el fotograma de inicio es mayor al fotograma final, está claro que los datos aportados por el usuario a Elesys no son lógicos, por lo que el comportamiento del programa puede ser imprevisible y no dar resultados útiles.

Por todo esto, es obligado un control estricto de la introducción de datos a través de la interfaz, tanto mediante las herramientas que MEL proporciona al crear los

elementos de la GUI como a través del código de las operaciones que utilizarán esos datos.

Conclusiones y Trabajos Futuros

Una vez finalizado todas las etapas de desarrollo, implementación y testeo de Elesys, en este último capítulo estamos preparados para abordar las conclusiones sobre el trabajo realizado.

A medida que se superan problemas y surgen otros nuevos durante la realización de Elesys, se modifica la visión, que en un principio se tiene, sobre la programación, los gráficos por ordenador, la computación científica, los métodos de elementos de contornos y elementos finitos, y otro temas abordados en este proyecto. Con la experiencia adquirida, se puede aportar un concepto más cercano a la realidad de lo que ha supuesto desarrollar Elesys.

Así mismo, es importante realizar un examen personal, tratando de analizar objetivamente, si se han alcanzado las expectativas y retos que en un principio se planteó cuando se decide esta materia como Proyecto Fin de Carrera.

Durante los meses de trabajo en este proyecto, es inevitable que a medida que se profundizaba en la realización del proyecto, se considerarán posibles objetivos

futuros aún más ambiciosos. Está claro, que estos otros proyectos relacionados con Elesys, exigen de una dedicación, tiempo y esfuerzo tan grande, que se consideró excesivo ser añadidos a este proyecto, pues provocaría, entre otros efectos, una desviación de los objetivos iniciales.

Sin embargo, estas metas suponen un estimulante campo de estudio y propone retos excitantes para cualquier persona que pueda continuar el camino que Elesys ha comenzado.

1. Consecución de objetivos.

Si se recuerda lo expuesto en el primer capítulo de esta memoria, los objetivos marcados eran los siguientes:

- Lectura, traducción y representación gráfica en tres dimensiones de mallas de elementos mediante la lectura de un fichero de datos proporcionado.
- Lectura, traducción y representación de la solución de la/s variable/s de las que conste el problema (desplazamientos, tensiones, temperatura, etc.) y conforme a resultados matemáticos almacenados en un fichero externo.
- Representación gráfica de la evolución temporal de la soluciones numéricas del problema en cuestión.
- Operaciones de representación que permitan el análisis en profundidad de los resultados gráficos obtenidos.

No sólo se observa que Elesys cumple cada uno de estos objetivos, sino que, en la mayoría de los casos, los supera y aporta nuevas funcionalidades no planteadas en un principio:

- La lectura, traducción y representación gráfica de las mallas es realizado por Elesys a una velocidad casi instantánea. No existen tiempo de esperas,

ni tiempo de carga. La representación gráfica presenta gran dinamismo y es manejada por la computadora de forma ágil y rápida.

- Elesys es capaz de leer y representar cualquier variable escalar o vectorial que se le aporte. Mediante una escala de colores o por el movimiento de la malla, en caso de hablar de la variable desplazamiento, el usuario puede ver la progresión de los valores de la variable en el modelo.

- Lo comentado hasta ahora es extrapolable al caso de la evolución temporal de la soluciones numéricas. Basta con modificar adecuadamente el formato del fichero para que Elesys lo trate con la misma eficacia que en el caso de representación por frecuencias.

- Antisimetrías, simetrías, representación por color por vertex, ocultación de elementos, suavizados, etc. son sólo algunas de las operaciones que soporta cualquier problema representado por Elesys.

2. Conclusiones

A medida que se avanzaba en el desarrollo del programa y se tenía mayor experiencia, se fue aceptando una serie de conclusiones y conocimientos que sería importante resaltar.

En primer lugar, hay que comprender que la industria del diseño por ordenador consta de una serie de peculiaridades que la hacen única. Tratar de crear un buen producto sin tener en cuenta estas particularidades, sólo conduce al fracaso del proyecto.

Igualmente, hay que asumir las limitaciones que una industria tan potente, compleja y que evoluciona a un ritmo tan frenético, impone. Frente a esto, sólo cabe dos caminos:

1. Luchar contra estas limitaciones, tratando de romperlas.
2. Asumirlas, tratar de aprovechar esta corriente de conocimiento y tecnología, y amoldarse al mercado.

Si se toma el primer camino, casi se puede asegurar que, cada vez que se supere algún obstáculo, surgirán otros muchos nuevos. Eso sin contar que, el obstáculo superado habrá sido ya solucionado por otros mucho antes y de una forma mejor que la elegida por el investigador.

Los resultados serán más provechosos a partir del momento que se tome el segundo camino y el investigador se amolde, conozca su sitio en el mercado y se coloque en un lugar provechoso que le permita desarrollar soluciones que funcionen eficientemente, aprovechando lo ya investigado por terceras partes.

Por último, hay que destacar que, aunque superficialmente pueda parecer extraño, la línea que separa computación y la representación gráfica es extremadamente delgada y, en la mayoría de los casos, se entremezclan, de forma que difícilmente es apreciable.

Un usuario inexperto puede pensar que ambas cosas son muy diferentes. Puede suponer que el campo de los gráficos por ordenador sólo se ocupan de mostrar información geométrica, puntos, colores y líneas por pantalla. Por otro lado, se puede creer que la computación sólo se ocupa de realizar cálculos matemáticos, resolver ecuaciones y procesar algoritmos. Sin embargo, en el campo que Elesys se desarrolla, ambas disciplinas están muy relacionadas.

Los gráficos por ordenador son el resultado de complejas operaciones matemáticas. Sin ellas, sería imposible mostrar ni siquiera la figura más simple imaginable. Igualmente, la geometría limita y guía las operaciones a computar por el ordenador. Métodos como el de elemento finito y elementos de contorno así lo muestran.

3. Líneas futuras de Investigación

Elesys comienza un camino con el proceso de postprocesado de problemas mediante el software infográfico Maya. Como se observa con su funcionamiento, la potencia gráfica de Maya bien encauzada puede aportar resultados muy prometedores y altamente útiles.

Sería muy interesante la realización de plug-in que resuelvan las dos etapas precedentes: el preprocesado y el proceso. El proceso, puede resultar la parte más alejada de las ventajas de la infografía. Su implementación realizada por este camino supone el mayor reto a conseguir.

Por el contrario, el preprocesado, presenta un campo apasionante para desarrollar en futuros proyectos cuyo resultados puede ser altamente productivos. La potencia de modelado de Maya puede suponer una ventaja competitiva inigualable por otras soluciones del mercado.

Entrando en otras líneas, sin alejarnos tanto del campo de estudio de Elesys, es también muy interesante el campo de suavizado de mallas. En Elesys se realiza un proceso útil del mismo. Sin embargo, un amplio estudio del mismo, puede conducir a economizar muchas horas de cálculos mediante la obtención de refinamiento de mallas por interpolación por splines u otros métodos alternativos. Esto permitiría trabajar con mallas preprocesadas de forma más "basta" que posteriormente podría refinarse mediante los resultados obtenidos en la etapa de procesado.

ANEXOS

Principios de Programación en Maya

1. Características de Programación de Maya.

Customización.

Toda la interfaz gráfica de usuario (GUI) de Maya, es escrita y controlada usando MEL (*Maya Embedded Language*). La creación, edición y eliminación de todos los elementos de la interfaz gráfica de usuario es realizada usando el lenguaje MEL. De hecho, se puede reemplazar completamente la interfaz estándar de Maya creando nuevos scripts MEL.

Además de la interfaz de usuario, se puede configurar las preferencias internas de Maya. Usando MEL se pueden hacer cambios en las preferencias de Maya, en ciertas partes determinadas o en una extensa parte del sistema. Por ejemplo, se puede asegurar que todos los usuarios usen las mismas unidades de forma consistente en tiempo, ángulos y distancias. Cualquier usuario que abra un proyecto podrá tener establecida una configuración concreta en ese instante.

Integración.

Normalmente, Maya no es el único paquete utilizado en un proyecto. Puede usarse sólo algunas características mientras que se usan otros paquetes para tareas específicas. Por tanto, hay una necesidad de exportar e importar datos de Maya a estos paquetes externos.

Aunque Maya viene con las características estándar de exportación e importación de ciertos formatos de datos, la interfaz de programación permite escribir exportadores e importadores de datos personalizados. Estos se conocen como traductores, traducen los datos de un paquete para que sea entendible por otro. Puesto que Maya proporciona el acceso completo a la escena y todos sus datos, se puede extraer estos datos de la manera que se necesite. Un requerimiento común para las compañías de juegos, es utilizar las ventajas que aporta Maya (modelos, animación, etcétera) y convertirlos a una forma que sea fácilmente utilizable por el motor gráfico del juego.

Las funciones de Maya también pueden ser compiladas en una aplicación autónoma. De esta manera, puede utilizar una combinación de Maya y otro código de programación para proporcionar una solución completa a la aplicación.

Automatización

A menudo, ciertas tareas son repetidas continuamente. Programar es una buena forma de automatizar las tareas repetitivas. Antes de tener al usuario repitiendo a mano una tarea una y otra vez, la escritura de un script MEL puede automatizar completamente el proceso.

Si hablamos de una aplicación de una red de sombreado, de posicionamiento de objetos, o simplemente de alguna tarea que es realizada más de una vez, ésta puede ser automatizada. De hecho, con la generalización que la programación

ofrece, se puede tener un script que realice tareas diferentes dependiendo del contexto en el que se encuentre en cada momento.

Extensiones

Aunque Maya ostenta tener un extenso conjunto de características, siempre habrá necesidad de herramientas y características a medida. Maya permite agregar su propias características y funciones. Estas características pueden estar hechas para trabajar continuamente con el resto de Maya. Desde el punto de vista del usuario, las herramientas que se crean no aparecen como herramientas distintas de las herramientas estándar de Maya. Usando MEL y C++ se puede crear un gran número de extensiones al paquete de Maya.

Conceptos Fundamentales de Maya

El enfoque y la filosofía de trabajo del software de Autodesk difiere en el camino y normas generales que pueden encontrarse en paquetes infográficos semejantes. En este capítulo se presenta una amplia descripción sobre la manera de trabajar internamente de Maya siendo esencial para la compresión del sistema Elesys.

Algunos sistemas imponen una serie de limitaciones sobre lo que se puede y no se puede hacer. Aunque Maya es, sin duda, muy flexible, hay algunas reglas fundamentales. De esta forma, es importante entender que Maya permite “hacer cosas de manera equivocada". Puesto que el entorno de trabajo de Maya es muy flexible, es raro que coloque pautas estrictas sobre cómo se debe realizar una tarea en particular. Esto lo hace muy atractivo, puesto que cualquier problema puede ser abordado desde varios ángulos diferentes. La desventaja de esta flexibilidad es que a veces permite que un desarrollador mal informado cree aplicaciones que no se ajusten apropiadamente dentro del entorno de trabajo.

Comprendiendo esto se observa la importancia de contar con una base de conocimiento sólida con el fin de obtener los mejores resultados en el desarrollo

de Elesys. Solo de esta forma se podrá optimizar el funcionamiento del software a la hora de realizar las tareas a las que tendrá que hacer frente el programa.

2. Arquitectura de Maya

Aunque la mayoría de las aplicaciones 3D realizan las mismas tareas que Maya, el enfoque que aplican es muy diferente. Los pasos que se deben realizar en un paquete dado para completar una operación concreta, normalmente es única y, por tanto, impone una cierta estructura predefinida en su flujo de trabajo.

Por el contrario, Maya tiene flujo de trabajo muy flexible que no impone excesivas limitaciones. En la mayoría de ocasiones hay más de una forma de acercarse a un problema particular. Esto significa que se puede usar cualquier enfoque que convenga según las necesidades del usuario.

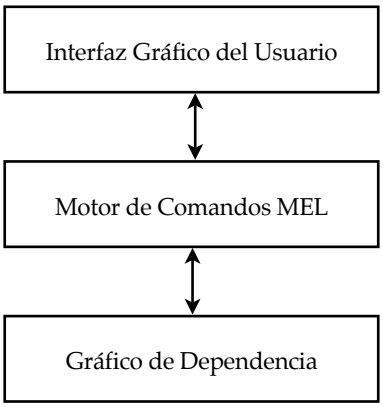


Figura A.1 Sistema Maya

Si tomamos el sistema completo de Maya y lo descomponemos en sus elementos básicos se tiene como resultado el esquema mostrado en la Figura A.1.

El usuario interactúa con el GUI de Maya: selecciona items de menú, cambia parámetros, anima y mueve objetos, etc. Cuándo se interactúa con el interfaz de usuario, Maya envía ordenes MEL al *Command Engine* (motor de comandos) donde son interpretadas y ejecutadas. Es posible ejecutar Maya en *batch mode* (modo tratamiento por lotes). En este modo no hay GUI, sino que las órdenes MEL son enviadas directamente al *Command Engine* para ser ejecutadas.

La mayoría de los comandos MEL operan con el *Dependency Graph* (Gráfico de la Dependencia). El *Dependency Graph* muestra de forma intuitiva la escena completa. La escena la componen todos los datos de la información importante

que forma el mundo 3D, incluyendo objetos, animación, dinámicas, materiales, etcétera. El *Dependency Graph* no sólo define qué datos están en la escena, sino su estructura y disposición estableciendo cómo son procesados. El *Dependency Graph* es el corazón y el cerebro de Maya.

Para obtener una mejor comprensión de las diferencias entre la arquitectura de Maya y las otras aplicaciones 3D, es importante ver cómo se diseñan la mayoría de las aplicaciones 3D "típicas". La comprensión del enfoque diferente de Maya indicará cuál es la mejor manera de diseñar *plugins*.

3. Enfoque tradicional

Generalmente, una aplicación 3D proporciona un conjunto de herramientas para realizar tareas de modelado, animación, iluminación y renderizado. Es evidente, que estas tareas son muy distintas unas de otras. Las operaciones que se realizan cuando se modela son muy diferentes de las que se hacen cuando se renderiza. También es muy distinta la información y los datos creados, manipulados y posteriormente almacenados, cuando se modela, de los que se tiene cuando se renderiza.

Puesto que cada área tiene su propio conjunto de datos y operaciones, es lógico que cada área pueda ser diseñada e implementada de forma separada. Cada área puede ser tratada en un módulo separado del resto. Así, cada módulo almacena y opera con un tipo de datos. El módulo de modelado, por ejemplo, operaría con datos geométricos y con datos de modelado. Se podría crear un módulo separado para cada una de las tareas: modelado, animación, iluminación y renderizado. Dentro de cada una de estas tareas, se aplicaría un conjunto definido de funciones. Así el resultado sería una interfaz de programación para cada módulo. Curiosamente, la misma claridad que proporciona esta interfaz, evita que sea flexible. Supongamos que se diseña una aplicación 3D que tiene un sistema de modelado. Una vez que ha modelado un objeto en su forma estática,

puede ser animado pasándolo al módulo de animación. El módulo de animación permite moverlo en el espacio. El módulo de renderizado podría después tomar los objetos animados desde el módulo de animación y renderizarlos en pantalla. Vemos que cada uno de los módulos tiene una interfaz concreta y sabe exactamente cómo comunicarse con los módulos de los que depende. Pero cabría preguntarse qué ocurre si se quisiera incluir un sistema de deformación que tome los objetos animados y los deforme. Puesto que el módulo de render sabe cómo comunicarse sólo con el módulo de animación, se tendrá que modificar y adaptar para incluir esta nueva utilidad o crear un nuevo módulo que permita realizar ésto. Si en algún momento necesitara añadir esta nueva función significaría que el módulo de animación y render necesitarían ser modificados. Por ello, la simple inclusión de una nueva característica podría tener un efecto rizado en el sistema completo dando lugar a muchas revisiones. La desventaja de este diseño es que no se permite agregar nuevas funciones sin hacer cambios en todos los módulos afectados.

4. Enfoque de Maya

En su nivel más esencial, una aplicación 3D crea datos que son después manipulados a través de una serie de operaciones. Por ejemplo, el resultado final de estos pasos de creación y manipulación de datos, podría ser una malla poligonal o una serie de píxeles en una imagen. El proceso de tomar unos datos, desde su origen hasta su resultado final, puede ser visto fácilmente como una serie de operaciones secuenciales. El resultado de una de estas operaciones alimenta a la siguiente. Cada operación modifica el dato de alguna forma, pasando el resultado a la siguiente operación que realiza otra modificación. De esta forma, el proceso completo puede ser entendido como una serie de datos que entran, y, al final de una serie de operaciones, sale otra serie de datos editados de una forma concreta. A esto se denomina normalmente *modelo de flujo*

de datos. En este modelo, los datos parecen estar fluyendo a través de las operaciones.

Cuando se considera lo que intenta conseguir una aplicación 3D, se observa que cada uno de los módulos se puede analizar como operaciones más pequeñas. Estas operaciones más pequeñas operan con los datos y después los pasa a la siguiente operación. Así, en un nivel muy esencial, todas las funciones proporcionadas por cada módulo pueden ser encapsuladas como una serie de operaciones interconectadas. Estas operaciones toman datos de entradas y producen datos de salida.

Conectando estos operadores de forma secuencial, se pueden considerar como un tubo o una tubería. Los datos entrarán por el primer operador de la tubería y posteriormente saldrán del último operador como datos procesados. Con el conjunto correcto de operadores, se puede colocar un modelo 3D en una de las salidas de las tuberías y acabar con una serie de píxeles en el otro extremo. Generalizando esto aún más, sencillamente se puede tener cualquier dato entrante y cualquier dato saliente. De hecho, no hay necesidad de imponer que los datos sean información 3D. Se podría colocar un fichero de texto como entrada y tener una serie de caracteres editados como salida.

Alias | Wavefront implementó el núcleo de Maya usando este paradigma de flujo de datos, incorporando el *Dependency Graph*. Por simplicidad, nos referiremos al *Dependency Graph* con la abreviatura DG. Antes de continuar, debemos destacar que, técnicamente, el DG está basado en un modelo de *push-pull* y no por un estricto modelo de flujo de datos. Explicaremos después, con más detalle, la distinción entre ambos pero, por ahora, el paradigma de flujo de datos nos proporcionará un entorno de trabajo intuitivo para comprender cómo funciona Maya.

El DG es realmente el corazón de Maya. Es el que proporciona todos los componentes constructivos mencionados. Además permite crear datos arbitrarios de entrada que se transformarán mediante una serie de operaciones para obtener resultados de datos procesados como salida. El dato y sus operaciones están encapsulados en el DG como nodos. Un nodo tiene un número de ranuras que contendrá datos usados por Maya. Los nodos además tienen un operador que permite que trabaje con estos datos para producir otros datos como resultado.

Las tareas más comunes en 3D pueden ser completadas mediante una serie de conexiones de nodos. Los datos del primer nodo serán la entrada del siguiente, y éste lo procesará de alguna forma o creará algún dato nuevo, que será la entrada del siguiente nodo. Así, los datos son modificados a través de la red de nodos desde el primero al último. A través de este camino, los nodos son libres de procesar y editar los datos del modo que a ellos les plazca. También pueden crear nuevos datos que, posteriormente, alimentará a otros nodos.

Cada tipo de nodos es diseñado para ofrecer sólo un pequeño y restringido conjunto de operaciones distintas. Cada uno proporciona sólo una función específica lo que hace que pueden quedar representados de forma más simple y manejable.

Para proporcionar modificaciones complejas de un dato, se puede crear una red. No hay restricción en el número de nodos que puede contener la red, ni en como se conecten. Así que del mismo modo que se pueden combinar una serie de bloques Lego para crear complejas estructuras, se pueden conectar nodos de Maya para generar complejos flujos de datos.

Una prueba de la flexibilidad del DG es que todos los datos (modelado, dinámicas, animación, sombreado, etc.) en Maya, son creados y manipulados a través de nodos. Este enfoque, basado en construir complejas estructuras a partir de bloques simples, da a Maya su verdadera potencia y flexibilidad. Mientras

que las aplicaciones de diseño 3D nombradas anteriormente usaban módulos separados, Maya proporciona una extensa serie de nodos distintos. Con un número suficientes de estos nodos conectados a una red, se puede completar casi cualquier tarea de computación. Todos las funciones de modelado, dinámicas, sombreado y render son dadas como una red de nodos interconectados. Analizada estas generalidades, se puede ver claramente como las funciones que se creen posteriormente pueden integrarse en este modelo más fácilmente, al no existir conjunto de interfaces ni módulos separados.

Para añadir una nueva funcionalidad a Maya, simplemente se necesita crear un nuevo nodo. De esta forma, al crearlos aparecerá un nodo nativo, es decir, se insertará un nuevo nodo en el DG. Esto, combinado con la habilidad de definir comandos propios, prácticamente permite posibilidades ilimitadas al extender Maya. Además, con el potente lenguaje script MEL, no hay prácticamente ningún aspecto de Maya que no se pueda ampliar o configurar a medida.

5. La Escena

La escena es el término común que describe el estado de una serie de gráficos 3D en una aplicación infográfica determinada. La escena comprende todos los modelos, sus animaciones, texturas, todas las luces, cámaras y demás información suplementaria, tales como opciones de render, etc.

En Maya la escena y el DG son una misma cosa. Cuando Maya almacena o carga la escena del disco, simplemente almacena o carga el DG. El DG define la escena completa a través de su red de nodos interconectados.

En una aplicación típica, toda la información que forma parte de la escena se almacena de forma separada en una estructura de datos concreta. Esta estructura normalmente reside en una base de datos central. Puesto que el DG es un conjunto de nodos interconectados, y esos nodos mantienen sus datos

internamente, la escena no es almacenada de forma centralizada sino como una red distribuida. Todos los nodos forman en su conjunto una base de datos distribuida. Y ya que se pueden añadir nuevos nodos y pueden eliminarse otros de la red, estamos ante una estructura altamente dinámica.

Con los datos siendo distribuidos a través de todos los nodos conectados, podría preguntarse cómo se recupera un dato determinado. Puesto que otros sistemas tienen los datos almacenados en una base de datos central, es fácil recuperar cierta información concreta. Es fácil, por ejemplo, pedir a la escena una lista de todas las luces existentes. Puesto que, en el caso de Maya, todos los datos están almacenados en una red distribuida de nodos, la recuperación de información de la escena podría pensarse que es difícil de realizar.

Para ello, Maya proporciona unas funciones determinadas para acceder a la escena como si fuera un almacén de datos. La interfaz de programación no obliga a conocer la disposición y estructura del DG. Por ejemplo, se puede obtener de forma rápida y fácil una lista de todos los objetos de un tipo dado sin necesidad de saber dónde están localizados en la red.

Aunque la interfaz de programación oculta toda la complejidad del procedimiento, es importante comprender qué ocurre internamente. El profundo conocimiento sobre cómo Maya almacena la escena permitirá desarrollar Elesys de forma más robustas y potente.

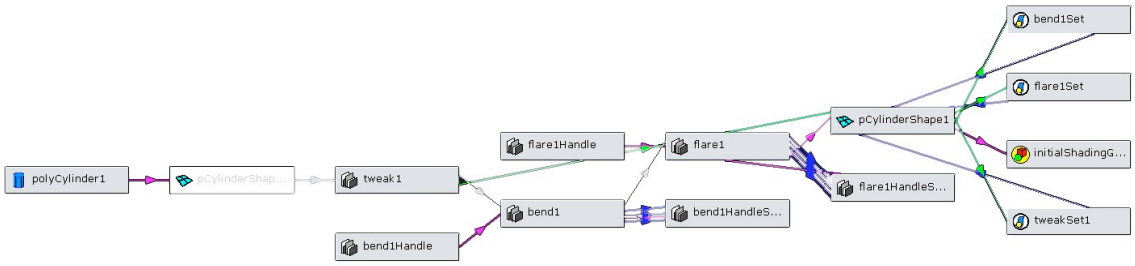


Figura A.2. Ejemplo del Dependency Graph

6. Visualizando el Dependency Graph

A continuación, hay una captura del *Hypergraph*. El *Hypergraph* es una ventana visual del DG, el núcleo de Maya. La Figura A.2. muestra el resultado de sólo algunas operaciones de modelado. Se puede ver que el número de nodos y sus conexiones se incrementan rápidamente después de sólo unas pocas operaciones.

Se puede observar que hay diferentes tipos de nodos: *joints*, *sets*, *tweaks*, etc. Estos nodos son conectados unos a otros. Las líneas muestran la conexiones entre nodos y la flechas muestran las direcciones que siguen los datos.

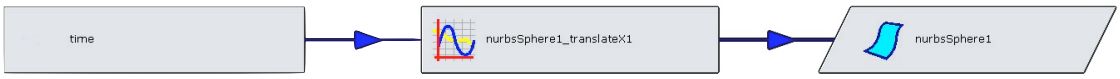


Figura A.3. Configuración típica del DG de animación

7. Flujo de Datos

Para un claro estudio del flujo de datos aislaremos una parte pequeña del DG y la estudiaremos en detalle. Mediante un ejemplo sencillo se analizará la forma de relacionarse y conectarse los datos de distintos nodos.

La Figura A.3. muestra un sencillo DG que presenta algunas de las funciones más comunes que se pueden ver en todas las redes. Los DG mayores y más complejos son el resultado de añadir más nodos y conexiones. El significado esencial de los nodos y conexiones permanece inalterables.

Esta configuración particular de nodos es muy común y muestra como normalmente se anima un valor en Maya. El DG de la figura está formado por sólo tres nodos. El primer nodo es un nodo de tiempo, el segundo es un nodo de *curveAnim* (curva de animación), y el nodo final es un nodo *transform* (tranformación).

Las líneas conectan los nodos y el flujo de información sigue la dirección indicada por las flechas. Puede verse que el nodo time1 se conecta con la entrada de la nurbsSphere_traslateX, que después se conecta con el nodo nurbsSphere1. La salida de time1 es simplemente el valor tiempo en ese instante. Por tanto, todas las escenas tiene un nodo time1. Dado el valor del tiempo en ese momento, la curva de animación genera un valor de salida. Este valor de salida es el resultado de evaluar la curva de animación para el valor del tiempo dado. Así pues, cuando el tiempo cambia, la curva de la animación se evalúa para distintos valores. Si la curva está variando, el resultado del valor también cambia, obteniéndose un valor animado siendo éste valor es la entrada traslateX del nodo transform nurbsSphere1. Este es la posición de los objetos a través del eje X. Como este valor varía la posición de nurbsSphere1 a lo largo del eje X también cambia.

Así, en resumen, la posición "x" de la esfera resulta de la evaluación de la curva de animación en el tiempo en ese momento. Un cambio en el tiempo actual causa que el nodo recalcule sus valores, y como resultado la esfera se mueva a lo largo del eje X. Se observa que se comienza en el primer nodo de la izquierda y se sigue procesándo a medida que va hacia la derecha hasta que llega al nodo final en la parte más alejada a la derecha. El flujo es esencialmente de izquierda a derecha en la dirección de las flechas conectadas.

8. Nodos

Puesto que los nodos son realmente los bloques constructivos fundamentales de Maya, es importante presentarlos con más detalle. Para comprender que contienen y como funcionan, serán presentados de forma esquemática. La Figura A.4 muestra un nodo genérico.

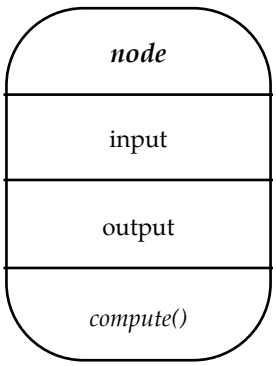


Figura A.4. Nodo Genérico

Cada nodo tiene una o más propiedades asociadas

que son normalmente conocidas como *atributos*. Un atributo es una propiedad particular de un nodo. Un nodo transform, por ejemplo, tiene un atributo translateX que contiene la posición actual del objeto a lo largo del eje X. En este ejemplo el nodo tiene dos atributos, input y output. Conceptualmente, cada nodo almacena el contenido de sus atributos internamente.

Además de los atributos, todos los nodos cuentan con una *función computación*. Este elemento no se muestra visualmente en Maya, sin embargo, hay que asumir que cada nodo posee uno. La función de computación tiene como tarea tomar uno o más atributos de entrada del nodo y, de una manera determinada previamente, transformarlos obteniendo uno o más atributos de salida. Como mencionamos antes, el principal objetivo de un nodo es almacenar datos y en ocasiones modificarlos de alguna forma. Es la función de computación la encargada de realizar la tarea de modificación de estos datos almacenados en los atributos. Para ello conoce cómo crear resultados como salida a partir los datos de entrada.

Entendiendo que la escena es realmente el DG, y que el DG es una serie de nodos interconectados, puede hacerse una interesante conclusión lógica sobre la función de computación. Las funciones de computación pueden considerarse las "neuronas" de la red de nodos que componen el cerebro de Maya. Mientras que en un cerebro humano son las neuronas las que se conectan en una red a través de las dendritas, en el cerebro de Maya son los nodos los que conectados entre sí y, a través de cada una de sus funciones de computación, causan que el flujo de datos se modifique de un nodo al siguiente.

La "inteligencia" de Maya es un resultado directo de la combinación de todas las funciones de computación. Puesto que el DG puede construirse a partir de muchos nodos diferentes, se puede crear distintos "cerebros" que creen y procesen datos. Mediante la creación de nuevos nodos con sus correspondientes funciones de computación, se puede añadir componentes propios dentro de esta

maquinaria. Los nodos creados por el desarrollador pueden ser añadidos a una o más zonas del DG. Esto puede formar funciones concretas de procesado de datos en general hechas a medida del usuario.

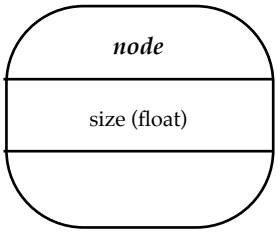


Figura A.5. Nodo Simple

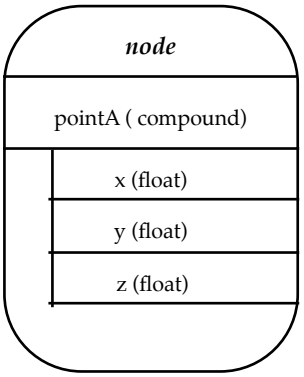


Figura A.6. Nodo con atributo complejo

9. Atributos

En los apartados anteriores hemos analizado cómo los datos de una escena son almacenados en los atributos de los nodos. Así un atributo es un almacén en el nodo que se ocupa de almacenar una información en particular. Cada atributo tendrá un nombre y guardará un tipo de dato determinado. El nombre es usado para proporcionar una descripción corta del atributo y es el que se utiliza para referenciarlo. Por ejemplo, el nombre radius es dado al atributo del nodo makeNurbsSphere para el control del radio de una esfera. Los tipos de datos del atributo son los que definen qué tipo de información se puede almacenar en él. Si, por ejemplo, un atributo es del tipo long, podrá almacenar un valor entero; Por otro lado, si es del tipo float, podrá almacenar el valor de una fracción. Estos dos tipos son referidos como *tipos de datos simples* puesto que pueden almacenar un solo valor individual. Una lista no exhaustiva de algunos de los tipos de datos simples son: boolean, byte, char, short, long, float, double, etc. Estos tipos corresponden a un tipo numérico básico disponible en la computadora. El nodo en la Figura A.5 tiene un solo atributo denominado size que es del tipo float.

Un atributo también puede almacenar *tipos de datos complejos*. Una malla geométrica completa o una superficie (NURBS) de B-splines racionales no uniforme pueden almacenarse como un atributo en un nodo. De hecho, todos los datos, sean simples o complejos, son almacenados como atributos. La habilidad de tener un atributo que cuente con cualquier dato de resultado, simple o complejo, es un gran convenio de flexibilidad.

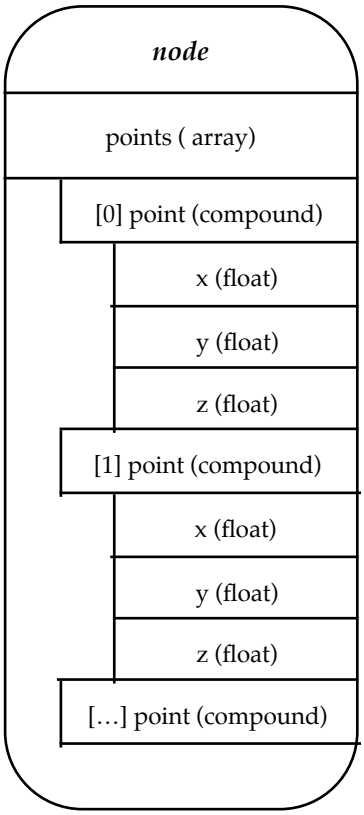


Figura A.7. Nodo con cadena de atributos complejos

Maya también le permite combinar los tipos de datos simples para formar estructuras más complejas. Por ejemplo, si se toma tres flotantes se puede definir una posición (point) en el espacio 3D. El punto podría consistir en tres atributos llamados x, y, z de tipo float. El punto puede ser considerado en sí mismo como el atributo padre de los atributos hijos x, y, z. En la Figura A.6 los nodos tienen un atributo pointA que consiste en tres atributos x, y, z. Maya se remite al atributo pointA como un atributo compuesto ya que es el resultado de combinar otros atributos. El atributo pointA es considerado padre mientras que los atributos x, y, z son considerados hijos. No hay limite en el número o el tipo de hijos que puede tener un atributo.

También se puede crear una cadena de atributos, que es, un atributo que contiene una secuencia de uno o más atributos. Por ejemplo, se puede querer almacenar una cadena de puntos que defina la posición de una serie de partículas. El nodo de la Figura A.7 cuenta con un atributo denominado point. Este atributo es una cadena de atributos punto. El atributo point es un atributo compuesto que contiene tres atributos hijos x, y, z. Este simple ejemplo crea una cantidad relativamente compleja y jerarquizada de atributos.

Así estamos ante una manera muy flexible de almacenar datos, pudiendo almacenarse datos simples o combinaciones más complejas de datos simples. Puesto que los atributos permiten crear una relación padre-hijo, el desarrollador es libre de crear jerarquías de arbitraria complejidad. Combinando esto con la posibilidad de almacenar cadenas de jerarquías, es posible representar complejas estructuras de datos usando atributos de Maya.

10. Función de Computación

Como mencionamos antes, la función de computación es el cerebro del nodo. Los atributos contienen los datos del nodo, pero no hacen nada con ellos. Para realizar algo útil, los datos del nodo deben ser procesados de alguna forma. Por ejemplo, para hacer que un personaje camine, tendría que moverse a medida que cambia el tiempo; los músculos y la piel del personaje tendría que cambiar. Comenzando por una pose estática del personaje, el sistema lo podría modificar y deformar a medida que el tiempo varía. En cada fotograma tendría que ser deformado de forma distinta. El resultado de la deformación en cada frame sería lo que se obtiene al poner al personaje estático a la entrada de una red DG de forma que se le tendría deformado a la salida de dicha red. En el nivel más básico, la función de computación tomará uno o más atributos de entrada y generará como resultado un atributo de salida. Como una función de programación, la función de computación puede ser considerada como sigue:

```
salida = computación ( entrada0 , … , entradaN )
```

Puesto que las entradas y las salidas son atributos y puesto que los atributos pueden tomar formas muy distintas, la función de computación puede operar con una gran variedad de datos. Puede por ejemplo tomar una superficie NURBS como entrada y generar una nueva superficie NURBS de salida. Podría tomar la superficie de entrada, aplicarle algún algoritmo obteniendo un valor de salida, por ejemplo, el área de una superficie. Por tanto, la flexibilidad aportada por la

generalidad del concepto de atributos hace que la función de computación pueda generar y tomar como entrada datos complejos arbitrarios.

Es muy importante destacar que los atributos de entrada y de salida que usa la función de computación son locales al nodo, es decir, que la función de computación atiende sólo a sus propios atributos a la hora de tomar y generar datos de salida; nunca toma información de otros nodos. Los datos que usa en sus cálculos deben venir de sí mismo y no de otro lado. Esto es una limitación muy importante a la que hay que atender ya que es fundamental para el diseño de nodos y para conseguir que éstos se comporten correctamente.

Internamente la función de computación es libre de hacer cualquier cosa que quiera para generar su resultado final. Las especificaciones exactas de cómo calcular la salida nunca son conocidas o expuestas en Maya. La función de computación puede ser considerada como una caja negra; no hay forma de determinar, a partir de su salida, cómo se ha llegado a ese resultado. La ventaja de este enfoque es que no se necesita saber como funciona el nodo internamente para utilizarlo. Un nodo que mezcla dos caras podría ser internamente muy complejo, pero nunca se tendrá que conocer. Cuando se use, todo lo que se necesita saber es que dado una serie de shape de entrada la función de computación genera un shape de salida.

Desde el punto de vista exterior, un nodo expone sólo el atributo que utiliza: su atributos de entrada y de salida. Estos atributos definen la *interfaz* del nodo. Cuando se use un nodo, se accederá a él mediante la obtención y determinación de sus atributos. Nunca se tendrá acceso directo a su función de computación. Un nodo makeNurbSphere genera una superficie NURBS con la forma de una esfera dada por sus atributos de entrada, radius, start sweep, etc. Se puede controlar cómo el nodo genera la superficie NURBS mediante el cambio de sus atributos de entrada. Para hacer la esfera mayor, se puede aumentar el valor de

su atributo radius. El nodo, posteriormente, genera una esfera mayor sin necesidad de conocer qué procesos realiza internamente para conseguirlo.

La ventaja de este diseño es que si, por ejemplo, se decide crear nuestro propio makeMyNurbsSphere seremos libres de implementar el método de generación de superficie como decidamos. Si posteriormente se crea un método más eficiente, es posible implementarlo sin romper la red DG. Lo único que ha importado es que la función de computación genera una superficie.

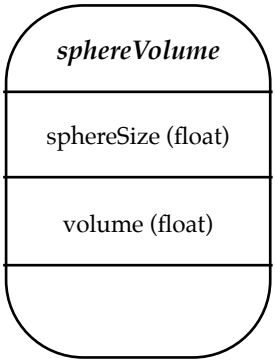


Figura A.8. Nodo Volumen Esfera

11. Atributos Dependientes

Es muy importante destacar que en la descripción de atributos en los apartados anteriores éstos se clasificaban en atributos de entrada o de salida. Maya no hace esta distinción. En Maya no hay atributos de entrada o de salida, todos los atributos sencillamente son almacenadores de datos.

Sin embargo, aunque Maya no realiza esta distinción, es necesario clasificar los atributos según sea entradas o salidas. Un atributo de entrada es aquel que alimenta la entrada de la función de computación de un nodo. Un atributo de salida es aquel que resulta de tomar uno o más atributos de entrada y calcular su valor usando la función de computación.

Para concretar se utilizará el nodo sphereVolume que se muestra en la Figura A.8. Tiene dos atributos, sphereSize y volume, cada uno de ellos almacenados como un valor float. Estos atributos simplemente contienen datos, no hay consideración de cuál es el atributo de entrada y cuál es el atributo de salida.

Se sabe que el volumen de una esfera depende de su tamaño, por lo tanto se puede deducir rápidamente que el valor del atributo volume debe depender del

valor del atributo sphereSize. El atributo volume puede ser considerado como el atributo de salida y el sphereSize como el atributo de entrada. El cálculo actual del valor atributo volume es responsabilidad de la función de computación.

El cálculo del volumen puede ser escrito esquemáticamente como sigue:

```
volume = computacion ( sphereSize )
```

Destacar que no se ha definido cómo se calcula el volumen.

Como dijimos antes, no es importante conocer como funciona internamente la función de computación. Para utilizar un nodo, solo se necesita saber qué atributo de entrada y de salida tiene. La función de computación se ocupa del resto.

A continuación se debe indicar las relaciones intrísecas entre ambos atributos. El atributo volume es el resultado que obtiene la función de computación al tomar como entrada la sphereSize. Si la sphereSize cambia, posteriormente el volume necesitará ser recalculado. Es necesario establecer un mecanismo para definir esta relación de dependencia entre la sphereSize y el atributo volume. Internamente, esto se realiza usando la función attributeAffects de la API de C++. Esta función dice a Maya que el atributo dado afecta a otro, indicando la relación de dependencia entre ambos atributos. En este ejemplo la función llamada podría tomar la siguiente forma:

```
attributeAffects ( sphereSize, volume );
```

Esto especifica que el atributo sphereSize afectará al valor del atributo volume. La llamada a esta función individual define dos importantes estados. Primero, informa a Maya que el volume es el resultado de un cálculo, es decir, que su valor siempre se obtendrá como resultado de un cálculo de la función de computación. Segundo, informa a Maya que la sphereSize debería ser usada como entrada de la función de conputación cuando el volume sea calculado.

Cada vez que se requiera el valor del atributo volume, se calcula el valor del sphereSize. Si este valor del atributo sphereSize se ha modificado se calculará de nuevo el valor del atributo volume.

Destacar que no se menciona en ningún momento cómo el nodo calcula el valor volumen. Internamente el nodo podría utilizar el algoritmo que se crea más adecuado para obtener este valor. El usuario puede requerir el valor del atributo volume, el nodo lo calculará devolviendo posteriormente el resultado.

Si se intenta programar un nodo, es fácil intuir dónde se dedicará más tiempo de desarrollo: escribiendo la función de computación. Aunque este simple ejemplo tiene un solo atributo de entrada que produce un solo atributo de salida, en la práctica la función de computación puede calcular un número cualquiera de atributos de salida a partir de un número cualquiera de atributos de entrada. Las relaciones de dependencias solamente tienen que ser definidas entre los atributos de entrada y los atributos de salida dados. Es también posible utilizar como entrada cualquier atributo de salida.

12. Tiempo

Es importante destacar que un nodo no tiene por qué calcular necesariamente algo. Es totalmente válido tener un nodo que solamente contenga valores en uno o más atributos. Estos nodos actúan como simples almacenadores de datos. El nodo tiempo es un ejemplo perfecto. El tiempo en Maya es almacenado en un nodo tiempo denominado time1 cuya disposición se muestra en la Figura A.9.

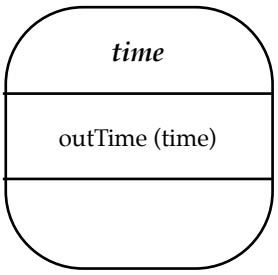


Figura A.9. Nodo Tiempo

El nodo tiempo solamente contiene un valor de tiempo en el atributo outTime. Cuando se mueve el deslizador de frame o se hace click en Play, Maya sitúa el

valor del frame en ese momento en el atributo outTime del nodo time1. El nodo no calcula nada aunque esto no significa que no contenga una función de computación. Todos los nodos tiene una función de computación. En este caso, puesto que no hay marcada ninguna relación de dependencia entre los atributos del nodo, la función de computación nunca es llamada.

13. Conectando Nodos

En apartados anteriores se ha definido qué es un nodo y cuales son sus características. Es evidente que un nodo aislado no proporciona mucha utilidad. La verdadera potencia real de los nodos es realizar complejos cálculos cuando se conectan en redes. Los nodos realizan cálculos según su función de computación pero ahora esos resultados serán entradas de otros nodos que posteriormente realizarán otros cálculos. Mediante la conexión de nodos, se pueden crear cadenas de cálculos computacionales con las que se obtendrán el resultado final, el cual será las salidas de los atributos de los nodos en cada uno de los finales de cadena.

Volviendo atrás, al ejemplo original de la animación DG, se puede indagar más profundamente y ver cómo cada nodo está conectado con el otro. La Figura A.10 muestra la configuración original de los nodos en el DG.

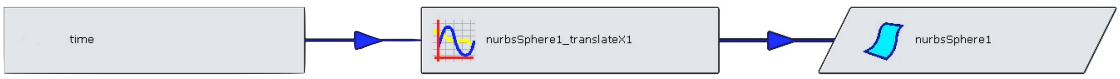


Figura A.10. Configuración de un nodo para animación.

Esquemáticamente este DG puede presentarse como se ve en la Figura A.11 observándose claramente los atributos de cada nodo y cómo se conectan. El nodo tiempo cuenta con un solo atributo de salida outTime, que indica el tiempo en dicho instante. Este valor de tiempo es el atributo de entrada del nodo animCurveTL. Este cuenta con una curva de animación la cual se puede editar en

el Graph Editor para animar un valor. Este nodo calcula el atributo de salida, entrada del atributo traslateX del nodo transform. El nodo transform es un nodo muy grande, con muchos atributos. Por simplicidad se muestran sólo los atributos traslate y sus atributos hijos, traslateX, traslateY, traslateZ.

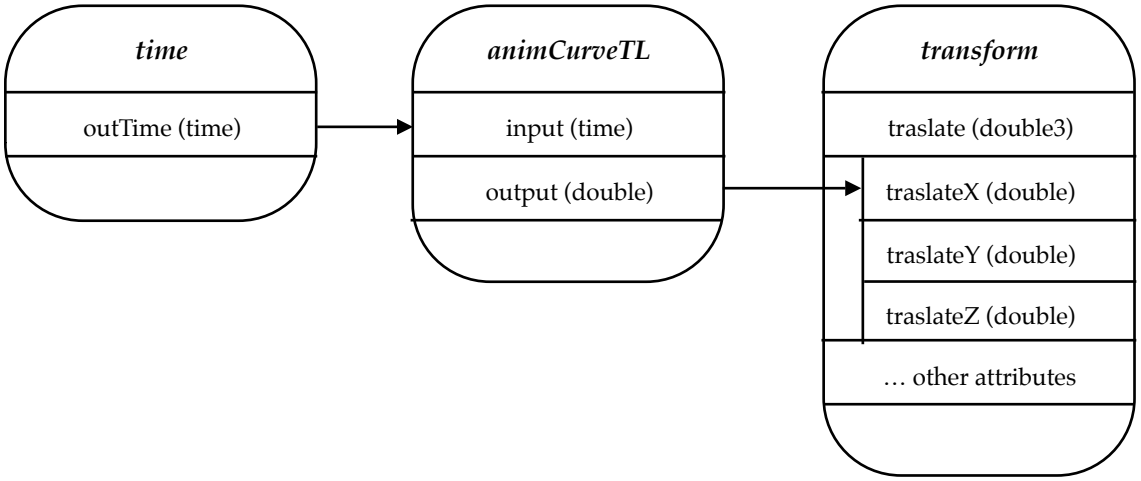


Figura A.11. Nodo con cadena de atributos complejos

Queda claro, por este diagrama, que el proceso de conectar nodos consiste en conectar los atributos de estos nodos. El flujo de datos y la información transcurre desde un atributo de un nodo a otro atributo de otro nodo. Destacar que un atributo puede conectarse sólo con otro atributo del mismo tipo. El atributo outTime es del tipo time. Es posible conectarlos al atributo de entrada del nodo animCurveTL ya que también es del tipo time. Sin embargo, no sería posible conectar el atributo outTime al traslateX del nodo transform puesto que es de un tipo distinto, double. Sin ser muy específico, Maya da mecanismos para convertir algunos tipos de datos concretos a otros tipos diferentes.

Es importante destacar que un nodo nunca es "consciente" de que está conectado a otro nodo o dónde se sitúa dentro de la red DG. De hecho un nodo solo es consciente de sus atributos de entrada y de salida. No conoce si están conectados o no. Maya conduce el flujo de datos desde un nodo al siguiente. Esta limitación a un conocimiento local limita a todos los nodos, permitiéndolos usar como

verdaderos bloques constructivos. Los nodos no sabrán nunca en que contexto han sido usados.

Un atributo puede conectarse con otros muchos atributos. Así el valor de ese atributo alimenta a todos los atributos a los que está conectado. Un ejemplo de esta configuración se muestra en la Figura A.12.

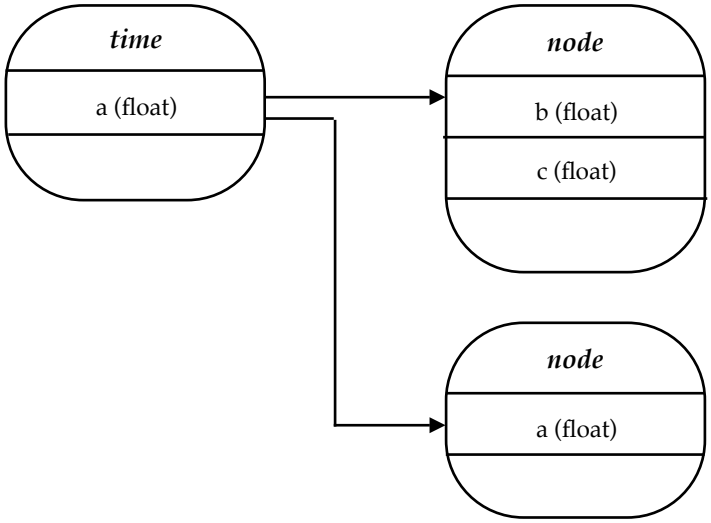


Figura A.12. Un solo atributo es conectado a múltiples destinos

No se permite, sin embargo, tener como entrada múltiples atributos como se muestra en la Figura A.13. Esta limitación es comprensible puesto que el atributo contiene un solo valor y si se le proporciona múltiples conexiones no podrá saber cuál de las conexiones usar para este valor. ¿Cuál es el valor de "a" en esta configuración? ¿Es b o c? No hay una respuesta correcta.

Cuando una conexión se hace para que un atributo origen conecte con otro atributo destino, el valor interno que tenía el atributo destino previamente es eliminado y toma el valor de la conexión entrante. Igualmente, cuando una conexión entre dos atributos se rompe, el valor almacenado en el que antes era el atributo destino es el último valor que él tenía antes de la desconexión. Si el valor que estaba alimentando al atributo conectado era 1,23, si después elimina la conexión, el atributo retiene el valor de 1,23. Puesto que ahora la conexión ha

sido rota, si el atributo origen cambia, el atributo aislado (antiguo atributo destino) no queda afectado.

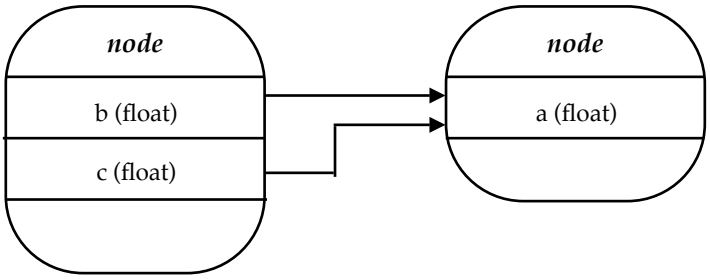


Figura A.13. Conexión Ilegal

14. Enfoque Push-Pull

Como mencionamos antes, el DG funciona, por lo menos en un sentido conceptual, como un modelo de flujo de datos. Aunque este modelo mental hace más fácil la comprensión de como los nodos se conectan para hacer que la máquina procese datos, realmente la implementación del DG es más sutil.

EL DG basa su funcionamiento en un modelo *push-pull* (empujar-extraer). Este modelo se define con la posibilidad de que cierta información pueda ser empujada (*pushed*) a través de la red mientras que otra información puede ser *extraída* (*pulled*). La diferencia entre las dos es que cuando se empuja un nodo, el efecto se propaga a todos los nodos conectados como salidas. Cuando se extrae información de un nodo, se propaga esa petición a través de todos los nodos que proporcionan conexión de entrada al nodo.

La necesidad de hacer esta distinción entre los dos modos de flujo de información se realiza por motivos de eficiencia. El modelo push-pull permite un mecanismo de actualización más eficiente que si se utilizara un modelo de flujo de datos puro.

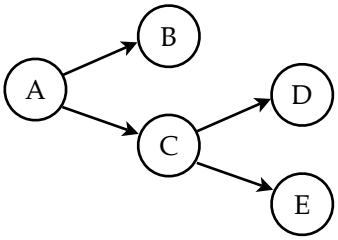


Figura A.14. Nodos Conectados

En la Figura A.14 hay cinco nodos interconectados. Cada uno de los nodos puede procesar datos que vayan a él, produciendo datos de resultados que son después su salida. En este ejemplo, el nodo **A** genera datos que después son pasados a su conexión de salida, nodos **B** y **C**. El nodo **C** no genera ningún dato directamente pero toma el dato de entrada del nodo **A** y éste es procesado antes de pasar el resultado a ambos nodos **D** y **E**.

Para demostrar la diferencia entre el enfoque de flujo de datos y el enfoque push-pull, se comenzará con algunos datos en el nodo **A** y se verá como se propaga a través de esta red de nodos simples.

Enfoque de flujo de datos.

Usando el enfoque de flujo de datos, se comienza por el nodo **A**, donde se generan algunos datos. éstos posteriormente alimentan la entrada de los nodos **B** y **C**, que después serán procesados. El nodo **C** pasa el resultado de este proceso al nodo **D** y **E**, que realizarán sus propios procesados. De esta descripción está claro que el dato comenzará desde el nodo más a la izquierda y se propagará a través de la red, con cada nodo haciendo un procesado de los datos a lo largo de este camino. Cuando el nodo **A** genera un nuevo dato, toda la red se actualiza. Cuando el proceso es completado, los nodos **B**, **D** y **E** cuentan con el resultado final.

Ahora imaginemos que el proceso hecho por cada nodo es extremadamente complejo. Digamos de forma exagerada, por ejemplo, que se toma una hora para cada proceso de cada nodo individual para procesar sus datos de entrada y posteriormente generar su dato de salida. Esto significa que si el nodo **A** genera algún dato nuevo, podría tomar un tiempo total combinado de cuatro horas antes de que todos los nodos (**B**, **D**, **E**) terminen de tener sus resultados.

Enfoque Push - Pull

¿Qué ocurre si, en lugar de querer los resultados de **B**, **D** y **E**, usted quiere sólo el resultado del nodo **B**? Desafortunadamente, bajo este modelo todos los datos se propagan a través de todos los nodos. Esto significa que aunque se pueda sólo querer el resultado del nodo **B**, se tendrá que esperar a todos los nodos hasta completar su procesado. La red de nodos que se ha presentado en este ejemplo está muy simplificada a propósito con fines aclaratorios pero se podría imaginar una red más compleja y los problemas de funcionamiento que este enfoque crearía.

Idealmente se podría seleccionar un nodo dado en la red y querer hacer la mínima cantidad de trabajo para actualizar ese nodo. Es en este caso donde el modelo push-pull es muy útil. Este modelo rompe la actualización de la red en dos pasos distintos: el primer paso es la propagación de un indicador de estado, y el segundo paso es el verdadero procesado de los datos. El primer paso es el empuje (*push*) y el segundo paso es la extracción (*pull*).

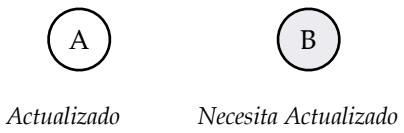


Figura A.15. Estado de los Nodos

Imagine que cada uno de los nodos cuenta con un indicador. Este indicador, al que se puede denominar indicador `needsUpdate`, indica que la salida del nodo no es válida durante más tiempo y que se necesita actualizar. Si el indicador está señalado, el nodo recalculará su salida. Si el indicador no esta señalado, pasará su valor de salida actual sin realizar ningún recálculo. El uso de este indicador trae una inmediata ganancia en velocidad. Cuando un nodo no necesita actualizar, todo su procesado se puede evitar y su resultado actual se pasa al siguiente nodo. Puesto que la actualización de la red sucede en dos fases, los nodos en el diagrama serán dibujados en uno de los dos estados: actualizado o

necesita actualizar. La Figura A.15 muestra como los nodos serán dibujados en sus distintos estados.

El primer paso del enfoque push-pull es el empuje a través de los indicadores estados a todos los nodos conectados. Cuando el nodo A genera un nuevo valor, los datos no se envían a los nodos conectados, pero su indicador de estado se propaga en su lugar. Este indicador de estado especifica cuál de los nodos necesita ahora actualizar. Los nodos **B** y **C** reciben su indicador y actualizan sus estados. Este paso continúa por todos los nodos. Finalmente, el nodo aparece en su nuevo estado como muestra la figura A.16.

Destacar que el nodo **A** no ha sido indicado como necesita actualizar, puesto que ya que él genera el cambio, contiene el valor más actualizado de los datos. Los demás datos cuentan con datos potencialmente antiguos ya que todos ellos dependen, directa o indirectamente del nodo **A**. Aunque que este paso no conlleva la realización de ningún proceso sobre los datos, esta parte se realiza muy rápidamente.

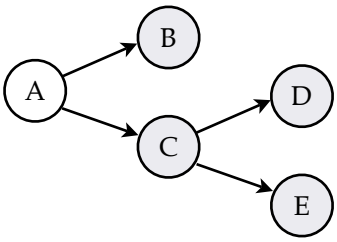


Figura A.16. Después de que el nodo A cambie

Llegados a este punto, los nodos permanecen sin cambios hasta que se realice una petición de los datos. ¿Qué ocurre si se quiere el resultado de salida del nodo **B**? Al obtener el resultado del nodo **B** se puede comparar con la extracción de información de dicho nodo. Así que el proceso de extracción es igual a preguntar al nodo sobre su salida.

Preguntar a un nodo determinado por su salida puede indirectamente conllevar la actualización de una serie de nodos. Todo ello para verificar que la entrada del nodo **B** está actualizada puesto que es la única manera de generar una salida correcta.

En primer lugar se chequea el indicador needsUpdate. Al estar activada, se sabe que la salida actual es antigua y que necesita ser recalculada. Así toma su entrada, en este caso la salida del nodo A, y recalcula su salida. Puesto que la salida esta actualizada, el indicador needsUpdate se apaga, indicando que el nodo contiene actualmente un dato de salida válido. Después de este paso de extracción, el gráfico aparece como muestra la Figura A.17.

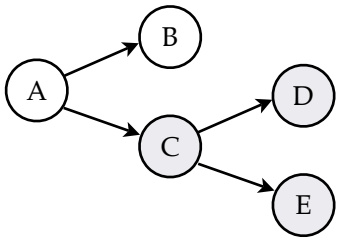


Figura A.17. Después de que el nodo B sea actualizado

Hay que destacar que sólo el nodo **B** ha sido actualizado. Como que sólo se hizo petición de su salida, sólo este nodo **B** fue actualizado. No se recalcularon los valores de los nodos **C**, **D** y **E** y su indicador needsUpdate seguirá encendido. El modelo *push-pull* permite elegir un solo nodo y pedir su resultado sin tener que actualizar los nodos no relacionados. Este enfoque de actualización solo de nodos que contribuyen a la salida final es lo que hace al modelo *push-pull* mucho más eficiente que el modelo de flujo de datos.

Ahora analizamos que ocurre si usted pide el resultado del nodo **D**. El mecanismo para su actualización es exactamente el mismo que para el nodo **B**, pero con alguna petición adicional. El paso en el que cada nodo realiza la actualización de sus datos es exactamente el mismo de antes. Se chequea si su indicador needsUpdate está encendido, y si lo está, genera una nueva salida. Si este indicador no está encendido, puede asumir de forma segura que su valor actual es el más reciente y actualizado, y sólo ha de pasar este valor. Este proceso se realiza igual en todos los nodos cuando se preguntan por su salida.

Cuando se pide la salida del nodo **D**, se ve que necesita actualización. Así **D** solicita su dato de entrada al nodo **C**. El nodo **C** ve igualmente que también necesita actualización, con lo pide su dato de entrada al nodo **A**. El nodo **A** no tiene su indicador needsUpdate encendido, por lo que sólo pasa su salida actual.

El nodo **C** toma su entrada para después actualizarse. Posteriormente apaga su indicador needsUpdate y pasa su nuevo resultado. El nodo **D** seguidamente toma su nuevo valor de entrada y calcula su nueva salida. Después su indicador needsUpdate también es restaurado. Al final del paso de empuje el gráfico aparece como se muestra en la Figura A.18.

Destacar que el nodo **E** no ha sido actualizado. Puesto que no está directa o indirectamente conectado al nodo **B**, no necesita preguntar sobre su actualización. Por otro lado, el nodo **C** ha sido actualizado puesto que está conectado al nodo **D** y necesita actualización.

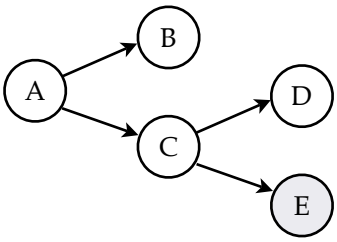


Figura A.18. Después de que el nodo D sea actualizado

Cuando la salida de un nodo es solicitada, el nodo solicita su entrada. Si la salida del nodo que le da la entrada necesita actualización, el proceso continúa. Esto puede provocar una cadena de actualizaciones en algunos o todos los nodos de la cadena.

Código Fuente

En las páginas siguientes se adjunta parte del código fuente de Elesys. Se ha obviado ciertas partes de código, como cabecera, registros, etc., por su carácter meramente formal y por carecer de interés informativo.

15. Generación de Maya

Genmalla.cpp[5]

```
//
// Copyright (C) 2008 Fernando Garcia Torcelly
//

#include <fstream.h>
#include <istream.h>
#include <regex.h>
#include <maya/MPxCommand.h>
#include <maya/MGlobal.h>
#include <maya/MFnPlugin.h>
#include <maya/MFnMesh.h>
#include <maya/MPoint.h>
#include <maya/MPointArray.h>
#include <maya/MArgList.h>
#include <maya/MSyntax.h>
#include <maya/MArgList.h>
```

[5] *El código fuente puede variar ligeramente entre las distintas plataformas en las que Elesys está desarrollada.*

```
class GenmallaCmd : public MPxCommand
{
public:
    virtual MStatus doIt(const MArgList&);
    static void *creator() { return new GenmallaCmd; }

};

MStatus GenmallaCmd::doIt(const MArgList &args)
{

    // Abre el fichero indicado
    MString myfile="Vacio";

    // Status
    MStatus stat;

    // Toma argumentos
    unsigned index;
    index = args.flagIndex( "f", "file" );
    if( MArgList::kInvalidArgIndex != index )
            args.get( index+1, myfile );

    // Si no se importa nombre del fichero
    if(myfile=="Vacio") {
            MGlobal::displayError(MString("Debe seleccionar un fichero
            de mallado"));
            return MS::kFailure;
    }

    // Abre el fichero
    ifstream filein;
    filein.open(myfile.asChar());

    // Si no se puede abrir el fichero
    if(!filein) {
            MGlobal::displayError(MString("No se ha podido abrir el
fichero"));
            return MS::kFailure;
    }

    // Fichero temporal de inicializacion
    MString myfile2="/Applications/Elesys.app/Contents/temporal/
    Inicio.dat";
    fstream fileout;
    fileout.open(myfile2.asChar(),ios::out);
    if(!fileout) {
```

```
                MGlobal::displayError(MString("No se ha podido abrir el
                fichero"));
                return MS::kFailure;
        }
        fileout<<"frecuencia|\n1|\n1|\n";

        // Inicializa el numero de puntos y numeros de elementos
        unsigned int NumPun;
        unsigned int NumEle;


        // Lee numero de lineas de puntos que tiene el fichero
        char NumLinPun[100];
        filein.getline(NumLinPun,100,'\n');


        // Comprueba que recoge entero
        regex_t retmp;
        regmatch_t mtmp;
        int res;

        // Comprueba que la primera linea responde al formato
        res = regcomp(&retmp, "^[0-9]+[\x0D]*$|^[0-9]+[\x0D]*[\|]{1}$",
REG_EXTENDED);
        res = regexec(&retmp, NumLinPun, (size_t) 1, &mtmp, 0);
        if (res!=0) {
                MGlobal::displayError(MString("El n° de puntos indicado no
                es un n° valido: ") + NumLinPun);
                return MS::kFailure;
        }

        // Hace cast de char a int
        NumPun=atoi(NumLinPun);


        // Lee posiciones de los puntos y almacena en matriz auxiliar
        char Puntos[200];
        char *PtoSep;
        int i,j;
        int seis,nueve;
        double PunAux[NumPun][5];

        // Inicializa el indice maximo del nodo del fichero
        int MaxID=0;

        for(i=0,j=0;i<=(NumPun-1);i++)
        {
                filein.getline(Puntos,200,'\n');
```

```
PtoSep=strtok(Puntos," |\t");

// Comprueba que es un numero flotante o entero
res = regcomp(&retmp, "^[\-]*[0-9]+[\x0D]*$|^[\-]*[0-9]+
[\x0D]*[\|]{1}$|^[\-]*[0-9]*[\.]{1}[0-9]+[\x0D]*$|
^[\-]*[0-9]*[\.]{1}[0-9]+[\x0D]*[\|]{1}$|^[\-]*[0-9]*[\.]
{1}[0-9]+[eE]{1}[\+\-]{1}[0-9]+[\x0D]*$|^[\-]*[0-9]*[\.]
{1}[0-9]+[eE]{1}[\+\-]{1}[0-9]+[\x0D]*[\|]{1}$",
REG_EXTENDED);
res = regexec(&retmp, PtoSep, (size_t) 1, &mtmp, 0);
if (res!=0) {
      int linea = i+2;
      int columna = j+1;
      MGlobal::displayError(MString("Error en puntos: linea
      ") + linea + MString(", columna ") + columna );
      return MS::kFailure;
}

// Pasa a flotante
PunAux[i][j]=atof(PtoSep);

if(i==0) {
      fileout<<PunAux[i][j]<<"|";
}

if(PunAux[i][j]>MaxID)
      MaxID=PunAux[i][j];

j++;

PtoSep=strtok(NULL," |\t");

while(PtoSep!=NULL)
{
      // Comprueba que es un numero flotante o entero
      res = regcomp(&retmp, "^[\-]*[0-9]+[\x0D]*$|
      ^[\-]*[0-9]+[\x0D]*[\|]{1}$|^[\-]*[0-9]*[\.]{1}[0-9]+
      [\x0D]*$|^[\-]*[0-9]*[\.]{1}[0-9]+[\x0D]*[\|]{1}$|
      ^[\-]*[0-9]*[\.]{1}[0-9]+[eE]{1}[\+\-]{1}[0-9]+
      [\x0D]*$|^[\-]*[0-9]*[\.]{1}[0-9]+[eE]{1}[\+\-]{1}
      [0-9]+[\x0D]*[\|]{1}$", REG_EXTENDED);
      res = regexec(&retmp, PtoSep, (size_t) 1, &mtmp, 0);
      if (res!=0) {
            int linea = i+2;
            int columna = j+1;
            MGlobal::displayError(MString("Nodos: linea ")
            + linea + MString(", columna ") + columna );
            return MS::kFailure;
```

```
                }


                PunAux[i][j]=atof(PtoSep);

                if(i==0) {
                        fileout<<PunAux[i][j]<<"|";
                }

                PtoSep=strtok(NULL," |\t");
                j++;
        }

        // Comprueba que el numero de columnas es correcto
        if (j!=4) {
                int index=j;
                int linea=i+1;
                MGlobal::displayError(MString("Nodos: La linea ") +
                linea + MString(" tiene ") + index + MString("
                columnas (distinto de 4) "));
                return MS::kFailure;
        }

        if(i==0) {
                fileout<<"0|0|0|0|0|0|";
        }

        j=0;
}

// Inicializa matriz de puntos final
double Pun[MaxID+1][5];
unsigned int IDAux;

for(i=0,j=0;i<=(NumPun-1);i++)
{
        //  Se pone en la posición Pun[IDAux][0]
        IDAux=static_cast<int>(PunAux[i][0]);
        //  Dice que, ejemplo, Pun[15][0]=15;
        Pun[IDAux][j]=PunAux[i][j];
        j++;
        while(j<=3)
        {
                //  Dice que, ejemplo, Pun[15][1]=...;Pun[15][2]=
                ...;Pun[15][3]=...;
                Pun[IDAux][j]=PunAux[i][j];
                j++;
        }
```

```
        Pun[IDAux][4]=i;
        j=0;
}


// Lee Numero de Elementos
char NumLinEle[100];
filein.getline(NumLinEle,100,'\n');


// Comprueba que el nº de elementos es valido
res = regcomp(&retmp, "^[0-9]+[\x0D]*$|^[0-9]+[\x0D]*[\|]{1}$",
REG_EXTENDED);
res = regexec(&retmp, NumLinEle, (size_t) 1, &mtmp, 0);
if (res!=0) {
        MGlobal::displayError(MString("El nº de elementos indicado
        no es un nº valido: ") + NumLinEle);
        return MS::kFailure;
}


// Hace un cast a entero
NumEle=atoi(NumLinEle);


// Lee Puntos de cada elemento
char Elementos[400];
char *EleSep;
double Ele[NumEle][11];


for(i=0,j=0,seis=0,nueve=0;i<=NumEle-1;i++)
{
        filein.getline(Elementos,200,'\n');
        EleSep=strtok(Elementos," |\t");

        // Comprueba que es un numero entero
        res = regcomp(&retmp, "^[0-9]+[\x0D]*$|^[0-9]+[\x0D]*[\|]
        {1}$", REG_EXTENDED);
        res = regexec(&retmp, EleSep, (size_t) 1, &mtmp, 0);
        if (res!=0) {
              int linea = i+3+NumPun;
              int columna = j+1;
              MGlobal::displayError(MString("Elementos: linea ") +
              linea + MString(", columna ") + columna );
              return MS::kFailure;
        }
```

```
		Ele[i][j]=atof(EleSep);
		j++;

		EleSep=strtok(NULL," |\t");

		while(EleSep!=NULL)
		{
			// Comprueba que es un numero entero
			res = regcomp(&retmp, "^[0-9]+[\x0D]*$|^[0-9]+
[\x0D]*[\|]{1}$", REG_EXTENDED);
			res = regexec(&retmp, EleSep, (size_t) 1, &mtmp, 0);
			if (res!=0) {
				int linea = i+3+NumPun;
				int columna = j+1;
				MGlobal::displayError(MString("Elementos: linea
") + linea + MString(", columna ") + columna );
				return MS::kFailure;
			}

			Ele[i][j]=atof(EleSep);
			EleSep=strtok(NULL," |\t");
			j++;
		}

		if(Ele[i][1]==9) {
			nueve++;

			// Comprueba cantidad de columnas
			if (j!=11) {
				int index=j;
				int linea=i+1;
				MGlobal::displayError(MString("Elementos: La
				linea ") + linea + MString(" tiene ") + index +
				MString(" columnas (distinto de 11) "));
				return MS::kFailure;
			}
		}
		else if(Ele[i][1]==6) {
			seis++;

			// Comprueba cantidad de columnas
			if (j!=8) {
				int index=j;
				int linea=i+1;
				MGlobal::displayError(MString("Elementos: La
				linea ") + linea + MString(" tiene ") + index +
				MString(" columnas (distinto de 8) "));
```

```
                        return MS::kFailure;
                }
        }


        // Reinicia las columnas
        j=0;
}


// 3° Parametro a recoger
// Numero de elementos en el array (Numero de nodos)
unsigned int count=NumPun;

// Ordena vertices del mallado en array
double scr[NumPun][4];
for(i=0,j=1;i<=(NumPun-1);i++)
{
        IDAux=PunAux[i][j-1];
        while(j<=3)
        {
        scr[i][j-1]=Pun[IDAux][j];
        j++;
        }
        scr[i][3]=1;
        j=1;
}

// Crea el array de puntos
MPointArray arrayPuntos(scr,count);


//  4° Parametro a recoger
// Poligonos: 8 por cada elemento
int NumPoligonos=(nueve*8)+(seis*4);
// Recoge en un array el numero de vertices de cada poligono
int scrint1[NumPoligonos];
        i=0;
        while(i<=(NumPoligonos-1))
        {
              scrint1[i]=3;
              i++;
        }

// Numero de numero poligonos de la malla actual
unsigned int countint1=NumPoligonos;

//  Crea el array de poligonos
```

```
MIntArray PolyCounts(scrint1,countint1);



//  5° Parametro a recoger
int AuxEleID;
//  Numero de conexiones en la malla (24 en cada elemento)
unsigned int countint2=(seis*12)+(nueve*24);
//  Se ordena como se conectan los vertices
int scrint2[countint2];i=0;j=0;

        while(j<=(NumEle-1))
        {
              if(Ele[j][1]==9) {
                    AuxEleID=static_cast<int>(Ele[j][10]);
                    scrint2[i+0]=Pun[AuxEleID][4];
                    AuxEleID=static_cast<int>(Ele[j][8]);
                    scrint2[i+1]=Pun[AuxEleID][4];
                    AuxEleID=static_cast<int>(Ele[j][9]);
                    scrint2[i+2]=Pun[AuxEleID][4];
                    AuxEleID=static_cast<int>(Ele[j][10]);
                    scrint2[i+3]=Pun[AuxEleID][4];
                    AuxEleID=static_cast<int>(Ele[j][9]);
                    scrint2[i+4]=Pun[AuxEleID][4];
                    AuxEleID=static_cast<int>(Ele[j][2]);
                    scrint2[i+5]=Pun[AuxEleID][4];
                    AuxEleID=static_cast<int>(Ele[j][10]);
                    scrint2[i+6]=Pun[AuxEleID][4];
                    AuxEleID=static_cast<int>(Ele[j][2]);
                    scrint2[i+7]=Pun[AuxEleID][4];
                    AuxEleID=static_cast<int>(Ele[j][3]);
                    scrint2[i+8]=Pun[AuxEleID][4];
                    AuxEleID=static_cast<int>(Ele[j][10]);
                    scrint2[i+9]=Pun[AuxEleID][4];
                    AuxEleID=static_cast<int>(Ele[j][3]);
                    scrint2[i+10]=Pun[AuxEleID][4];
                    AuxEleID=static_cast<int>(Ele[j][4]);
                    scrint2[i+11]=Pun[AuxEleID][4];
                    AuxEleID=static_cast<int>(Ele[j][10]);
                    scrint2[i+12]=Pun[AuxEleID][4];
                    AuxEleID=static_cast<int>(Ele[j][4]);
                    scrint2[i+13]=Pun[AuxEleID][4];
                    AuxEleID=static_cast<int>(Ele[j][5]);
                    scrint2[i+14]=Pun[AuxEleID][4];
                    AuxEleID=static_cast<int>(Ele[j][10]);
                    scrint2[i+15]=Pun[AuxEleID][4];
                    AuxEleID=static_cast<int>(Ele[j][5]);
                    scrint2[i+16]=Pun[AuxEleID][4];
```

```
                        AuxEleID=static_cast<int>(Ele[j][6]);
                        scrint2[i+17]=Pun[AuxEleID][4];
                        AuxEleID=static_cast<int>(Ele[j][10]);
                        scrint2[i+18]=Pun[AuxEleID][4];
                        AuxEleID=static_cast<int>(Ele[j][6]);
                        scrint2[i+19]=Pun[AuxEleID][4];
                        AuxEleID=static_cast<int>(Ele[j][7]);
                        scrint2[i+20]=Pun[AuxEleID][4];
                        AuxEleID=static_cast<int>(Ele[j][10]);
                        scrint2[i+21]=Pun[AuxEleID][4];
                        AuxEleID=static_cast<int>(Ele[j][7]);
                        scrint2[i+22]=Pun[AuxEleID][4];
                        AuxEleID=static_cast<int>(Ele[j][8]);
                        scrint2[i+23]=Pun[AuxEleID][4];
                        i=i+24;
                        j++;
                }
                else if(Ele[j][1]==6) {
                        AuxEleID=static_cast<int>(Ele[j][2]);
                        scrint2[i+0]=Pun[AuxEleID][4];
                        AuxEleID=static_cast<int>(Ele[j][7]);
                        scrint2[i+1]=Pun[AuxEleID][4];
                        AuxEleID=static_cast<int>(Ele[j][5]);
                        scrint2[i+2]=Pun[AuxEleID][4];
                        AuxEleID=static_cast<int>(Ele[j][5]);
                        scrint2[i+3]=Pun[AuxEleID][4];
                        AuxEleID=static_cast<int>(Ele[j][3]);
                        scrint2[i+4]=Pun[AuxEleID][4];
                        AuxEleID=static_cast<int>(Ele[j][6]);
                        scrint2[i+5]=Pun[AuxEleID][4];
                        AuxEleID=static_cast<int>(Ele[j][6]);
                        scrint2[i+6]=Pun[AuxEleID][4];
                        AuxEleID=static_cast<int>(Ele[j][5]);
                        scrint2[i+7]=Pun[AuxEleID][4];
                        AuxEleID=static_cast<int>(Ele[j][7]);
                        scrint2[i+8]=Pun[AuxEleID][4];
                        AuxEleID=static_cast<int>(Ele[j][7]);
                        scrint2[i+9]=Pun[AuxEleID][4];
                        AuxEleID=static_cast<int>(Ele[j][6]);
                        scrint2[i+10]=Pun[AuxEleID][4];
                        AuxEleID=static_cast<int>(Ele[j][4]);
                        scrint2[i+11]=Pun[AuxEleID][4];
                        i=i+12;
                        j++;
                }
        }

    //  Crea el array de poligonos
```

```
    MIntArray PolyConnects(scrint2,countint2);


    //  Inicializa variables necesaria
    MObject objTransform;
    MFnMesh meshFn;




    //  Crea la malla
    objTransform=meshFn.create(NumPun,NumPoligonos,arrayPuntos,
    PolyCounts,PolyConnects,MObject::kNullObj,&stat);
    if(!stat){
            stat.perror("Unable to create mesh");
            MGlobal::displayError(MString("Error desconocido: No se ha
            podido general el mallado."));
    }

    //  Se crea el sombreado del mallado
    meshFn.updateSurface();
    MString cmd("sets -e -fe initialShadingGroup ");
    cmd += meshFn.name();
    MGlobal::executeCommand(cmd);

    // Selecciona malla
    MString cmd1("select -r polySurface1");
    MGlobal::executeCommand(cmd1);

    // Escala
    MString cmd2("scale -r 0.1 0.1 0.1");
    MGlobal::executeCommand(cmd2);

    // Gira
    MString cmd3("rotate -r -os 0 0 0");
    MGlobal::executeCommand(cmd3);



    //  Finaliza el programa con exito
    return MS::kSuccess;
}



MStatus initializePlugin(MObject obj)
{
    MFnPlugin pluginFn(obj,"Fernando Garcia Torcelly","1.0");
```

```
    MStatus stat;
    stat=pluginFn.registerCommand("Genmalla",GenmallaCmd::creator);

    if(!stat)
            stat.perror("registerCommand failed");

    return stat;
}



MStatus uninitializePlugin(MObject obj)
{
    MFnPlugin pluginFn(obj);

    MStatus stat;
    stat=pluginFn.deregisterCommand("Genmalla");

    if(!stat)
            stat.perror("deregisterCommand failed");

    return stat;
}
```

16. Generación de caché

GencacheCmd.cpp

```
//
// Copyright (C) 2008 Fernando Garcia Torcelly
//

#include <fstream.h>
#include <istream.h>
#include <regex.h>
#include <maya/MPxCommand.h>
#include <maya/MGlobal.h>
#include <maya/MFnPlugin.h>
#include <maya/MFnMesh.h>
#include <maya/MPoint.h>
#include <maya/MPointArray.h>
#include <maya/MArgList.h>
#include <maya/MSyntax.h>
#include <maya/MArgList.h>
#include <maya/MFileObject.h>
```

```
class GenCacheCmd : public MPxCommand
{
public:
    virtual MStatus doIt(const MArgList&);
    static void *creator() { return new GenCacheCmd; }

};

MStatus GenCacheCmd::doIt(const MArgList &args)
{

    // Abre el fichero indicado
    MString myfile="Vacio";
    MString dirpath="Vacio";
    double Amplitud=1;
    int ejes=0;
    double Velocidad=1;

    // Status
    MStatus stat;

    unsigned index;
    index = args.flagIndex( "f", "file" );
    if( MArgList::kInvalidArgIndex != index )
            args.get( index+1, myfile );

    index = args.flagIndex( "d", "directory" );
    if( MArgList::kInvalidArgIndex != index )
            args.get( index+1, dirpath );

    index = args.flagIndex( "a", "amplitud" );
    if( MArgList::kInvalidArgIndex != index )
            args.get( index+1, Amplitud );

    index = args.flagIndex( "e", "ejes" );
    if( MArgList::kInvalidArgIndex != index )
            args.get( index+1, ejes );

    index = args.flagIndex( "v", "velocidad" );
    if( MArgList::kInvalidArgIndex != index )
            args.get( index+1, Velocidad );



    // Obtención del directorio
    const char *directorio=dirpath.asChar();
```

```
// Declaracion de variables para lectura de fichero de movimiento
float frec;
char NombreLec[300];
char Puntos[250];
char *PtoSep;
int frame;


// Si no se importa nombre del fichero
if(myfile=="Vacio") {
        MGlobal::displayError(MString("Debe seleccionar un fichero
        de mallado"));
        return MS::kFailure;
}


// Abre fichero de datos
fstream filein;
filein.open(myfile.asChar(),ios::in);


// Si no se puede abrir el fichero
if(!filein) {
        MGlobal::displayError(MString("No se ha podido abrir el
        fichero"));
        return MS::kFailure;
}

// Inicialización de variables para regex
regex_t retmp;
regmatch_t mtmp;
int res;


// Lee el tipo de variable
char Linea[100];
filein.getline(Linea,99,'\n');
PtoSep=strtok(Linea," |\t");


// Comprueba que la primera linea responde al formato (flotante)
res = regcomp(&retmp, "^temporal$|^frecuencia$|^temporal[\|]{1}$|
^frecuencia[\|]{1}$", REG_EXTENDED);
res = regexec(&retmp, PtoSep, (size_t) 1, &mtmp, 0);
if (res!=0) {
```

```
        MGlobal::displayError(MString("El tipo de fichero indicado
        no es valido: ") + PtoSep);
        return MS::kFailure;
}

// Guarda el tipo del fichero
bool temporal = false;
bool frecuencia = false;
res = regcomp(&retmp, "^temporal$|^temporal[\|]{1}$",
REG_EXTENDED);
res = regexec(&retmp, PtoSep, (size_t) 1, &mtmp, 0);
if (res!=0) {
        frecuencia=true;
}
else {
        temporal=true;
}
if(frecuencia&&temporal){
        MGlobal::displayError(MString("Error recogiendo tipo de
        fichero"));
        return MS::kFailure;
}


if(frecuencia) {
        // Lee la frecuencia que tiene el fichero
        filein.getline(Linea,99,'\n');
        PtoSep=strtok(Linea," |\t");


        // Comprueba que la primera linea responde al formato
        // (flotante)
        res = regcomp(&retmp, "^[0-9]+[\x0D]*$|^[0-9]+[\x0D]*[\|]
        {1}$|^[0-9]+[\.]{1}[0-9]+[\x0D]*$|^[0-9]+[\.]{1}[0-9]+
        [\x0D]*[\|]{1}$", REG_EXTENDED);
        res = regexec(&retmp, PtoSep, (size_t) 1, &mtmp, 0);
        if (res!=0) {
              MGlobal::displayError(MString("La frecuencia indicada
              no es valida: ") + PtoSep);
              return MS::kFailure;
        }


        // Hace un cast de char a float
        frec=atof(PtoSep);


        // Calculo de el numero de frame a calcular
```

```
int Numframes=240;

// Numeros de puntos a leer
unsigned int NumPun;
filein.getline(Linea,99,'\n');
PtoSep=strtok(Linea," |\t");


// Comprueba que la primera linea responde al formato
// (entero)
res = regcomp(&retmp, "^[0-9]+[\x0D]*$|^[0-9]+[\x0D]*[\|]
{1}$", REG_EXTENDED);
res = regexec(&retmp, PtoSep, (size_t) 1, &mtmp, 0);
if (res!=0) {
      MGlobal::displayError(MString("El n° de puntos
      indicado no es un n° valido: ") + PtoSep);
      return MS::kFailure;
}


// Hace cast de char a int
NumPun=atoi(PtoSep);


// Inicializa variables necesarias para recoleccion de
// datos y la matriz
float PunMov[NumPun][10];
float var;
int entero;
int i,j;

// Comienza a leer
for(i=0,j=0;i<=(NumPun-1);i++)
{
      filein.getline(Puntos,199,'\n');
      PtoSep=strtok(Puntos," |\t");


      // Comprueba que es un numero flotante (Exponencial o
      // no) o entero
      res = regcomp(&retmp, "^[\-]*[0-9]+[\x0D]*$|
      ^[\-]*[0-9]+[\x0D]*[\|]{1}$|^[\-]*[0-9]*[\.]{1}[0-9]+
      [\x0D]*$|^[\-]*[0-9]*[\.]{1}[0-9]+[\x0D]*[\|]{1}$|
      ^[\-]*[0-9]*[\.]{1}[0-9]+[eE]{1}[\+\-]{1}[0-9]+
      [\x0D]*$|^[\-]*[0-9]*[\.]{1}[0-9]+[eE]{1}[\+\-]{1}
      [0-9]+[\x0D]*[\|]{1}$", REG_EXTENDED);
      res = regexec(&retmp, PtoSep, (size_t) 1, &mtmp, 0);
      if (res!=0) {
```

```
        int linea = i+3;
        int columna = j+1;
        MGlobal::displayError(MString("Linea ") + linea
        + MString(", columna ") + columna );
        return MS::kFailure;
}


// Pasa a flotante
PunMov[i][j]=atof(PtoSep);
j++;

PtoSep=strtok(NULL," |\t");

while(PtoSep!=NULL)
{
        // Comprueba que es un numero flotante
        //(Exponencial o no) o entero
        res = regcomp(&retmp, "^[\-]*[0-9]+[\x0D]*$|
        ^[\-]*[0-9]+[\x0D]*[\|]{1}$|^[\-]*[0-9]*[\.]{1}
        [0-9]+[\x0D]*$|^[\-]*[0-9]*[\.]{1}[0-9]+
        [\x0D]*[\|]{1}$|^[\-]*[0-9]*[\.]{1}[0-9]+[eE]
        {1}[\+\-]{1}[0-9]+[\x0D]*$|^[\-]*[0-9]*[\.]{1}
        [0-9]+[eE]{1}[\+\-]{1}[0-9]+[\x0D]*[\|]{1}$",
        REG_EXTENDED);
        res = regexec(&retmp, PtoSep, (size_t) 1,
        &mtmp, 0);
        if (res!=0) {
                int linea = i+3;
                int columna = j+1;
                MGlobal::displayError(MString("Linea ") +
                linea + MString(", columna ") +
                columna );
                return MS::kFailure;
        }


        PunMov[i][j]=atof(PtoSep);
        PtoSep=strtok(NULL," |\t");
        j++;
}


// Comprueba que el numero de columnas es correcto
if (j!=10) {
        int index=j;
        int linea=i+1;
```

```
                MGlobal::displayError(MString("La linea ") +
                linea + MString(" tiene ") + index + MString("
                columnas (distinto de 10 Frec) "));
                return MS::kFailure;
        }

        j=0;
}

filein.close();

//  ----------------------------------------

// Inicializa ficheros
char NombreEsc[200];
FILE *FicheroLec;
FILE *FicheroEsc;



// Prepara cadena para invertir
unsigned char valor[4];
float SegFrame=0.04166666*float(Velocidad);
float tActual;
float ByteDesc=(NumPun*12)+52;
float Numbytes=NumPun*12;
char indice[10];



// Comienza para el frame 1
for(frame=1;frame<=Numframes;frame++)
{

        // Escribe float en ficheros para luego invertir
        strcpy(NombreEsc,directorio);
        strcat(NombreEsc,"aux.bin");
        ofstream fico1(NombreEsc,ios::binary);

        tActual=250*frame;

        // Escribe segun estandar de cache maya
        entero=1179603508;
        fico1.write((char*)&entero, sizeof(entero));

        entero=40;
        fico1.write((char*)&entero, sizeof(entero));
```

```
entero=1128350536;
fico1.write((char*)&entero, sizeof(entero));

entero=1448235854;
fico1.write((char*)&entero, sizeof(entero));

entero=4;
fico1.write((char*)&entero, sizeof(entero));

entero=808333568;
fico1.write((char*)&entero, sizeof(entero));

entero=1398032717;
fico1.write((char*)&entero, sizeof(entero));

entero=4;
fico1.write((char*)&entero, sizeof(entero));

entero=tActual;
fico1.write((char*)&entero, sizeof(entero));
entero=1163151693;
fico1.write((char*)&entero, sizeof(entero));
entero=4;
fico1.write((char*)&entero, sizeof(entero));
entero=tActual;
fico1.write((char*)&entero, sizeof(entero));

entero=1179603508;
fico1.write((char*)&entero, sizeof(entero));
entero=ByteDesc;
fico1.write((char*)&entero, sizeof(entero));
entero=1297695560;
fico1.write((char*)&entero, sizeof(entero));
entero=1128812109;
fico1.write((char*)&entero, sizeof(entero));

entero=18;
fico1.write((char*)&entero, sizeof(entero));
entero=1886350457;
fico1.write((char*)&entero, sizeof(entero));
entero=1400205926;
fico1.write((char*)&entero, sizeof(entero));
entero=1633903955;
fico1.write((char*)&entero, sizeof(entero));

entero=1751216229;
fico1.write((char*)&entero, sizeof(entero));
entero=822083584;
```

```
fico1.write((char*)&entero, sizeof(entero));
entero=1397316165;
fico1.write((char*)&entero, sizeof(entero));
entero=4;
fico1.write((char*)&entero, sizeof(entero));

fico1.write((char*)&NumPun, sizeof(NumPun));
entero=1180058433;
fico1.write((char*)&entero, sizeof(entero));
entero=Numbytes;
fico1.write((char*)&entero, sizeof(entero));


// Comienza a iterar y escribir en fichero binario
for(i=0;i<=NumPun;i++)
{
      switch(ejes)
      {
            case 0:     // Calculo de los tres ejes
                  var=PunMov[i][1]+(Amplitud*
                  (sqrt((PunMov[i][4]*PunMov[i][4])+
                  (PunMov[i][5]*PunMov[i][5])))*
                  (cos(frec*(SegFrame*frame))));
                  fico1.write((char*)&var,
                  sizeof(var));
                  var=PunMov[i][2]+(Amplitud*(
                  sqrt((PunMov[i][6]*PunMov[i][6])+
                  (PunMov[i][7]*PunMov[i][7])))*(
                  cos(frec*(SegFrame*frame))));
                  fico1.write((char*)&var,
                  sizeof(var));
                  var=PunMov[i][3]+(Amplitud*(
                  sqrt((PunMov[i][8]*PunMov[i][8])+
                  (PunMov[i][9]*PunMov[i][9])))*(
                  cos(frec*(SegFrame*frame))));
                  fico1.write((char*)&var,
                  sizeof(var));
                  break;

            case 1:     // Calculo en el eje x
                  var=PunMov[i][1]+(Amplitud*(
                  sqrt((PunMov[i][4]*PunMov[i][4])+
                  (PunMov[i][5]*PunMov[i][5])))*(
                  cos(frec*(SegFrame*frame))));
                  fico1.write((char*)&var,
                  sizeof(var));
                  var=PunMov[i][2];
```

```
        fico1.write((char*)&var,
        sizeof(var));
        var=PunMov[i][3];
        fico1.write((char*)&var,
        sizeof(var));
        break;

case 2:     // Calculo en el eje y
        var=PunMov[i][1];
        fico1.write((char*)&var,
        sizeof(var));
        var=PunMov[i][2]+(Amplitud*(
        sqrt((PunMov[i][6]*PunMov[i][6])+
        (PunMov[i][7]*PunMov[i][7])))*(
        cos(frec*(SegFrame*frame))));
        fico1.write((char*)&var,
        sizeof(var));
        var=PunMov[i][3];
        fico1.write((char*)&var,
        sizeof(var));
        break;

case 3:     // Calculo en el eje z
        var=PunMov[i][1];
        fico1.write((char*)&var,
        sizeof(var));
        var=PunMov[i][2];
        fico1.write((char*)&var,
        sizeof(var));
        var=PunMov[i][3]+(Amplitud*(
        sqrt((PunMov[i][8]*PunMov[i][8])+
        (PunMov[i][9]*PunMov[i][9])))*(
        cos(frec*(SegFrame*frame))));
        fico1.write((char*)&var,
        sizeof(var));
        break;

case 4:     // Calculo en el eje xy
        var=PunMov[i][1]+(Amplitud*(
        sqrt((PunMov[i][4]*PunMov[i][4])+
        (PunMov[i][5]*PunMov[i][5])))*(
        cos(frec*(SegFrame*frame))));
        fico1.write((char*)&var,
        sizeof(var));
        var=PunMov[i][2]+(Amplitud*(
        sqrt((PunMov[i][6]*PunMov[i][6])+
        (PunMov[i][7]*PunMov[i][7])))*(
        cos(frec*(SegFrame*frame))));
```

```
                fico1.write((char*)&var,
                sizeof(var));
                var=PunMov[i][3];
                fico1.write((char*)&var,
                sizeof(var));
                break;

            case 5:     // Calculo en el eje xz
                var=PunMov[i][1]+(Amplitud*(
                sqrt((PunMov[i][4]*PunMov[i][4])+
                (PunMov[i][5]*PunMov[i][5])))*(
                cos(frec*(SegFrame*frame))));
                fico1.write((char*)&var,
                sizeof(var));
                var=PunMov[i][2];
                fico1.write((char*)&var,
                sizeof(var));
                var=PunMov[i][3]+(Amplitud*(
                sqrt((PunMov[i][8]*PunMov[i][8])+
                (PunMov[i][9]*PunMov[i][9])))*(
                cos(frec*(SegFrame*frame))));
                fico1.write((char*)&var,
                sizeof(var));
                break;

            case 6:     // Calculo en el eje yz
                var=PunMov[i][1];
                fico1.write((char*)&var,
                sizeof(var));
                var=PunMov[i][2]+(Amplitud*(
                sqrt((PunMov[i][6]*PunMov[i][6])+
                (PunMov[i][7]*PunMov[i][7])))*(
                cos(frec*(SegFrame*frame))));
                fico1.write((char*)&var,
                sizeof(var));
                var=PunMov[i][3]+(Amplitud*(
                sqrt((PunMov[i][8]*PunMov[i][8])+
                (PunMov[i][9]*PunMov[i][9])))*(
                cos(frec*(SegFrame*frame))));
                fico1.write((char*)&var,
                sizeof(var));
                break;

            default : // Por defecto en los tres ejes
                var=PunMov[i][1]+(Amplitud*(
                sqrt((PunMov[i][4]*PunMov[i][4])+
                (PunMov[i][5]*PunMov[i][5])))*(
                cos(frec*(SegFrame*frame))));
```

```
                                    fico1.write((char*)&var,
                                    sizeof(var));
                                    var=PunMov[i][2]+(Amplitud*(
                                    sqrt((PunMov[i][6]*PunMov[i][6])+
                                    (PunMov[i][7]*PunMov[i][7])))*(
                                    cos(frec*(SegFrame*frame))));
                                    fico1.write((char*)&var,
                                    sizeof(var));
                                    var=PunMov[i][3]+(Amplitud*(
                                    sqrt((PunMov[i][8]*PunMov[i][8])+
                                    (PunMov[i][9]*PunMov[i][9])))*(
                                    cos(frec*(SegFrame*frame))));
                                    fico1.write((char*)&var,
                                    sizeof(var));
                                    break;
                        }
                  }
                  fico1.close();


                  // Seteamos los ficheros de lectura y escritura
                  // Para escribir el binario invertido
                  strcpy(NombreLec,directorio);
                  strcat(NombreLec,"aux.bin");

                  sprintf(indice,"%d",frame);
                  strcpy(NombreEsc,directorio);
                  strcat(NombreEsc,"polySurfaceShape1Frame");
                  strcat(NombreEsc,indice);
                  strcat(NombreEsc,".mc");

                  FicheroLec = fopen(NombreLec,"r");
                  FicheroEsc = fopen(NombreEsc,"w");


                  for(i=0;i<((NumPun*3)+27);i++)
                  {
                        fscanf( FicheroLec, "%c%c%c%c", &valor[0],
                        &valor[1], &valor[2], &valor[3] );
                        fprintf( FicheroEsc, "%c%c%c%c", valor[3],
                        valor[2], valor[1], valor[0] );
                  }

                  fclose(FicheroEsc);

fclose(FicheroLec);

            }
```

```
            // Escribe el fichero XML
            // Tiempo final en tricks
            int Numfinal=Numframes*250;

            char filexml[200];
            strcpy(filexml,directorio);
            strcat(filexml,"polySurfaceShape1.xml");

            FILE *fichero;
            fichero = fopen(filexml,"w");
            fprintf(fichero, "<?xml version=\"1.0\"?>\n<Autodesk
            _Cache_File>\n  <cacheType Type=\"OneFilePerFrame\"/>
            <time Range=\"250-%d\"/>\n  <cacheTimePerFrame TimePer
            Frame=\"250\"/>\n  <cacheVersion Version=\"2.0\"/>\n
            <extra>untitled</extra>\n  <extra>maya 8.5</extra>\n
            <extra>Fernando</extra>\n  <Channels>\n    <channel0
            ChannelName=\"polySurfaceShape1\" ChannelType=
            \"FloatVectorArray\" ChannelInterpretation=\"positions\"
            SamplingType=\"Regular\" SamplingRate=\"250\" StartTime=
            \"250\" EndTime=\"%d\"/>\n  </Channels>\n</
            Autodesk_Cache_File>", Numfinal, Numfinal );
            fclose(fichero);
            }


    if(temporal) {
            // Numeros de puntos a leer
            unsigned int NumPun;
            filein.getline(Linea,99,'\n');
            PtoSep=strtok(Linea," |\t");


            // Comprueba que la primera linea responde al formato
(entero)
            res = regcomp(&retmp, "^[0-9]+[\x0D]*$|^[0-9]+[\x0D]*[\|]
            {1}$", REG_EXTENDED);
            res = regexec(&retmp, PtoSep, (size_t) 1, &mtmp, 0);
            if (res!=0) {
                  MGlobal::displayError(MString("El nº de puntos
                  indicado no es un nº valido: ") + PtoSep);
                  return MS::kFailure;
            }

            // Hace cast de char a int
            NumPun=atoi(PtoSep);

            // Lee el numero de frames a representar
```

```
filein.getline(Linea,99,'\n');
PtoSep=strtok(Linea," |\t");


// Comprueba que la primera linea responde al formato
(entero)
res = regcomp(&retmp, "^[0-9]+[\x0D]*$|^[0-9]+[\x0D]*[\|]
{1}$", REG_EXTENDED);
res = regexec(&retmp, PtoSep, (size_t) 1, &mtmp, 0);
if (res!=0) {
      MGlobal::displayError(MString("El número de puntos no
      es valido: ") + PtoSep);
      return MS::kFailure;
}


// Calculo de el numero de frame a calcular
int Numframes=atoi(PtoSep);


// Inicializa variables necesarias para recoleccion de
// datos y la matriz
float PunMov[NumPun][4];
float PunEst[NumPun][4];
float var;
int entero;
int i,j,k;


// Inicializa ficheros
char NombreEsc[200];
FILE *FicheroLec;
FILE *FicheroEsc;


// Prepara cadena para invertir
unsigned char valor[4];
float SegFrame=0.04166666*float(Velocidad);
float tActual;
float ByteDesc=(NumPun*12)+52;
float Numbytes=NumPun*12;
char indice[10];


// Lee el primer grupo de posiciones que es para el modelo
// sin deformar
for(i=0,j=0;i<=(NumPun-1);i++)
{
```

```
filein.getline(Puntos,199,'\n');
PtoSep=strtok(Puntos," |\t");

// Comprueba que es un numero flotante (Exponencial o
//no) o entero
res = regcomp(&retmp, "^[\-]*[0-9]+[\x0D]*$|
^[\-]*[0-9]+[\x0D]*[\|]{1}$|^[\-]*[0-9]*[\.]{1}[0-9]+
[\x0D]*$|^[\-]*[0-9]*[\.]{1}[0-9]+[\x0D]*[\|]{1}$|
^[\-]*[0-9]*[\.]{1}[0-9]+[eE]{1}[\+\-]{1}[0-9]+
[\x0D]*$|^[\-]*[0-9]*[\.]{1}[0-9]+[eE]{1}[\+\-]{1}
[0-9]+[\x0D]*[\|]{1}$", REG_EXTENDED);
res = regexec(&retmp, PtoSep, (size_t) 1, &mtmp, 0);
if (res!=0) {
      int linea = i+3;
      int columna = j+1;
      MGlobal::displayError(MString("Linea ") + linea
      + MString(", columna ") + columna );
      return MS::kFailure;
}

// Pasa a flotante
PunEst[i][j]=atof(PtoSep);
j++;

PtoSep=strtok(NULL," |\t");

while(PtoSep!=NULL)
{
      // Comprueba que es un numero flotante
      // (Exponencial o no) o entero
      res = regcomp(&retmp, "^[\-]*[0-9]+[\x0D]*$|
      ^[\-]*[0-9]+[\x0D]*[\|]{1}$|^[\-]*[0-9]*[\.]{1}
      [0-9]+[\x0D]*$|^[\-]*[0-9]*[\.]{1}[0-9]+
      [\x0D]*[\|]{1}$|^[\-]*[0-9]*[\.]{1}[0-9]+[eE]
      {1}[\+\-]{1}[0-9]+[\x0D]*$|^[\-]*[0-9]*[\.]{1}
      [0-9]+[eE]{1}[\+\-]{1}[0-9]+[\x0D]*[\|]{1}$",
      REG_EXTENDED);
      res = regexec(&retmp, PtoSep, (size_t) 1,
      &mtmp, 0);/*[\+]{1}[0-9]+*/
      if (res!=0) {
            int linea = i+3;
            int columna = j+1;
            MGlobal::displayError(MString("Linea ") +
            linea + MString(", columna ") +
            columna );
            return MS::kFailure;
      }
```

```
                PunEst[i][j]=atof(PtoSep);
                PtoSep=strtok(NULL," |\t");
                j++;
        }



        // Comprueba que el numero de columnas es correcto
        if (j!=4) {
                int index=j;
                int linea=i+1;
                MGlobal::displayError(MString("La linea ") +
                linea + MString(" tiene ") + index + MString("
                columnas (distinto de 4) "));
                return MS::kFailure;
        }

        j=0;
}



// Comienza a leer
for(k=1;k<=Numframes;k++)
{
        for(i=0,j=0;i<=(NumPun-1);i++)
        {
                filein.getline(Puntos,199,'\n');
                PtoSep=strtok(Puntos," |\t");

                // Comprueba que es un numero flotante
                // (Exponencial o no) o entero
                res = regcomp(&retmp, "^[\-]*[0-9]+[\x0D]*$|
                ^[\-]*[0-9]+[\x0D]*[\|]{1}$|^[\-]*[0-9]*[\.]{1}
                [0-9]+[\x0D]*$|^[\-]*[0-9]*[\.]{1}[0-9]+
                [\x0D]*[\|]{1}$|^[\-]*[0-9]*[\.]{1}[0-9]+[eE]
                {1}[\+\-]{1}[0-9]+[\x0D]*$|^[\-]*[0-9]*[\.]{1}
                [0-9]+[eE]{1}[\+\-]{1}[0-9]+[\x0D]*[\|]{1}$",
                REG_EXTENDED);
                res = regexec(&retmp, PtoSep, (size_t) 1,
                &mtmp, 0);
                if (res!=0) {
                        int linea = i+3;
                        int columna = j+1;
                        MGlobal::displayError(MString("Linea ") +
                        linea + MString(", columna ") +
                        columna );
```

```
        return MS::kFailure;
}

// Pasa a flotante
PunMov[i][j]=atof(PtoSep);
j++;

PtoSep=strtok(NULL," |\t");

while(PtoSep!=NULL)
{
// Comprueba que es un numero flotante
// (Exponencial o no) o entero
        res = regcomp(&retmp, "^[\-]*[0-9]+
        [\x0D]*$|^[\-]*[0-9]+[\x0D]*[\|]{1}$|
        ^[\-]*[0-9]*[\.]{1}[0-9]+[\x0D]*$|
        ^[\-]*[0-9]*[\.]{1}[0-9]+[\x0D]*[\|]{1}$|
        ^[\-]*[0-9]*[\.]{1}[0-9]+[eE]{1}[\+\-]{1}
        [0-9]+[\x0D]*$|^[\-]*[0-9]*[\.]{1}[0-9]+
        [eE]{1}[\+\-]{1}[0-9]+[\x0D]*[\|]{1}$",
        REG_EXTENDED);
        res = regexec(&retmp, PtoSep, (size_t) 1,
        &mtmp, 0);
        if (res!=0) {
              int linea = i+3;
              int columna = j+1;
              MGlobal::displayError(
              MString("Linea ") + linea +
              MString(", columna ") + columna );
              return MS::kFailure;
        }


        PunMov[i][j]=atof(PtoSep);
        PtoSep=strtok(NULL," |\t");
        j++;
}


// Comprueba que el numero de columnas es
// correcto
if (j!=4) {
        int index=j;
        int linea=i+1;
        MGlobal::displayError(MString("La linea
        ") + linea + MString(" tiene ") + index +
        MString(" columnas (distinto de 4) "));
        return MS::kFailure;
```

```
}

j=0;

// Escribe float en ficheros para luego
// invertir
strcpy(NombreEsc,directorio);
strcat(NombreEsc,"aux.bin");
ofstream fico1(NombreEsc,ios::binary);

tActual=250*frame;

// Escribe segun estandar de cache maya
entero=1179603508;
fico1.write((char*)&entero, sizeof(entero));
entero=40;
fico1.write((char*)&entero,
sizeof(entero));
entero=1128350536;
fico1.write((char*)&entero,
sizeof(entero));
entero=1448235854;
fico1.write((char*)&entero, sizeof(entero));

entero=4;
fico1.write((char*)&entero, sizeof(entero));
entero=808333568;
fico1.write((char*)&entero,
sizeof(entero));
entero=1398032717;
fico1.write((char*)&entero,
sizeof(entero));
entero=4;
fico1.write((char*)&entero, sizeof(entero));

entero=tActual;
fico1.write((char*)&entero, sizeof(entero));
entero=1163151693;
fico1.write((char*)&entero,
sizeof(entero));
entero=4;
fico1.write((char*)&entero,
sizeof(entero));
entero=tActual;
fico1.write((char*)&entero, sizeof(entero));

entero=1179603508;
fico1.write((char*)&entero, sizeof(entero));
```

```
entero=ByteDesc;
fico1.write((char*)&entero,
sizeof(entero));
entero=1297695560;
fico1.write((char*)&entero,
sizeof(entero));
entero=1128812109;
fico1.write((char*)&entero, sizeof(entero));

entero=18;
fico1.write((char*)&entero, sizeof(entero));
entero=1886350457;
fico1.write((char*)&entero,
sizeof(entero));
entero=1400205926;
fico1.write((char*)&entero,
sizeof(entero));
entero=1633903955;
fico1.write((char*)&entero,
sizeof(entero));

entero=1751216229;
fico1.write((char*)&entero, sizeof(entero));
entero=822083584;
fico1.write((char*)&entero,
sizeof(entero));
entero=1397316165;
fico1.write((char*)&entero,
sizeof(entero));
entero=4;
fico1.write((char*)&entero,
sizeof(entero));

fico1.write((char*)&NumPun, sizeof(NumPun));
entero=1180058433;
fico1.write((char*)&entero,
sizeof(entero));
entero=Numbytes;
fico1.write((char*)&entero, sizeof(entero));


// Comienza a iterar y escribir en fichero
// binario
for(i=0;i<=NumPun;i++)
{
      switch(ejes)
      {
```

```
case 0:     // Calculo de los tres
            // ejes
      var=PunMov[i][1];
      fico1.write((char*)&var,
      sizeof(var));
      var=PunMov[i][2];
      fico1.write((char*)&var,
      sizeof(var));
      var=PunMov[i][3];
      fico1.write((char*)&var,
      sizeof(var));
      break;

case 1:     // Calculo en el eje x
      var=PunMov[i][1];
      fico1.write((char*)&var,
      sizeof(var));
      var=PunEst[i][2];
      fico1.write((char*)&var,
      sizeof(var));
      var=PunEst[i][3];
      fico1.write((char*)&var,
      sizeof(var));
      break;

case 2:     // Calculo en el eje y
      var=PunEst[i][1];
      fico1.write((char*)&var,
      sizeof(var));
      var=PunMov[i][2];
      fico1.write((char*)&var,
      sizeof(var));
      var=PunEst[i][3];
      fico1.write((char*)&var,
      sizeof(var));
      break;

case 3:     // Calculo en el eje z
      var=PunEst[i][1];
      fico1.write((char*)&var,
      sizeof(var));
      var=PunEst[i][2];
      fico1.write((char*)&var,
      sizeof(var));
      var=PunMov[i][3];
      fico1.write((char*)&var,
      sizeof(var));
      break;
```

```
case 4:     // Calculo en el eje xy
      var=PunMov[i][1];
      fico1.write((char*)&var,
      sizeof(var));
      var=PunMov[i][2];
      fico1.write((char*)&var,
      sizeof(var));
      var=PunEst[i][3];
      fico1.write((char*)&var,
      sizeof(var));
      break;

case 5:     // Calculo en el eje xz
      var=PunMov[i][1];
      fico1.write((char*)&var,
      sizeof(var));
      var=PunEst[i][2];
      fico1.write((char*)&var,
      sizeof(var));
      var=PunMov[i][3];
      fico1.write((char*)&var,
      sizeof(var));
      break;

case 6:     // Calculo en el eje yz
      var=PunEst[i][1];
      fico1.write((char*)&var,
      sizeof(var));
      var=PunMov[i][2];
      fico1.write((char*)&var,
      sizeof(var));
      var=PunMov[i][3];
      fico1.write((char*)&var,
      sizeof(var));
      break;

default : // Por defecto en los
            tres ejes
      var=PunMov[i][1];
      fico1.write((char*)&var,
      sizeof(var));
      var=PunMov[i][2];
      fico1.write((char*)&var,
      sizeof(var));
      var=PunMov[i][3];
      fico1.write((char*)&var,
      sizeof(var));
```

```
                                    break;

                              }
                        }
                        fico1.close();



                        // Seteamos los ficheros de lectura y escritura
                        // Para escribir el binario invertido
                        strcpy(NombreLec,directorio);
                        strcat(NombreLec,"aux.bin");

                        sprintf(indice,"%d",frame);
                        strcpy(NombreEsc,directorio);
                        strcat(NombreEsc,"polySurfaceShape1Frame");
                        strcat(NombreEsc,indice);
                        strcat(NombreEsc,".mc");

                        FicheroLec = fopen(NombreLec,"r");
                        FicheroEsc = fopen(NombreEsc,"w");


                        for(i=0;i<((NumPun*3)+27);i++)
                        {
                              fscanf( FicheroLec, "%c%c%c%c",
                              &valor[0], &valor[1], &valor[2],
                              &valor[3] );
                              fprintf( FicheroEsc, "%c%c%c%c",
                              valor[3], valor[2], valor[1], valor[0] );
                        }

                        fclose(FicheroEsc);


fclose(FicheroLec);

                  }


                  }

            // Escribe el fichero XML
            // Tiempo final en tricks
            int Numfinal=Numframes*250;

            char filexml[200];
            strcpy(filexml,directorio);
```

```
            strcat(filexml,"polySurfaceShape1.xml");

            FILE *fichero;
            fichero = fopen(filexml,"w");
            fprintf(fichero, "<?xml version=\"1.0\"?>\n<Autodesk_
            Cache_File>\n  <cacheType Type=\"OneFilePerFrame\"/>
            <time Range=\"250-%d\"/>\n  <cacheTimePerFrame
            TimePerFrame=\"250\"/>\n  <cacheVersion Version=\"2.0\"/>
            \n  <extra>untitled</extra>\n  <extra>maya 8.5</extra>\n
            <extra>Fernando</extra>\n  <Channels>\n    <channel0
            ChannelName=\"polySurfaceShape1\" ChannelType=
            \"FloatVectorArray\" ChannelInterpretation=\"positions\"
            SamplingType=\"Regular\" SamplingRate=\"250\" StartTime=
            \"250\" EndTime=\"%d\"/>\n  </Channels>\n</
            Autodesk_Cache_File>", Numfinal, Numfinal );
            fclose(fichero);

            }

            filein.close();


            //  ----------------------------------------

    return MS::kSuccess;
}



MStatus initializePlugin(MObject obj)
{
    MFnPlugin pluginFn(obj,"Fernando Garcia Torcelly","1.0");

    MStatus stat;
    stat=pluginFn.registerCommand("Genmov",GenCacheCmd::creator);

    if(!stat)
            stat.perror("registerCommand failed");

    return stat;
}



MStatus uninitializePlugin(MObject obj)
{
    MFnPlugin pluginFn(obj);
```

```
    MStatus stat;
    stat=pluginFn.deregisterCommand("Genmov");

    if(!stat)
            stat.perror("deregisterCommand failed");

    return stat;
}
```

17. Antisimetría

pluginMain.cpp

```
//
// Copyright (C) 2008 Fernando Garcia Torcelly
//

#include "AntiSimetriaNode.h"
#include "AntiSimetriaCmd.h"
#include <maya/MFnPlugin.h>


MStatus initializePlugin( MObject obj )
{
    MStatus stat;
    MString errStr;
    MFnPlugin plugin( obj, "Fernando Garcia Torcelly", "1.0", "Any");

    stat = plugin.registerCommand( "antisimetria",
    AntiSimetriaCmd::creator );
    if ( !stat )
    {
            errStr = "registerCommand failed";
            goto error;
    }

    stat = plugin.registerNode( "antisimetria", AntiSimetriaNode::id,
    AntiSimetriaNode::creator, AntiSimetriaNode::initialize );
    if ( !stat )
    {
            errStr = "registerNode failed";
            goto error;
    }

    return stat;

error:

    stat.perror( errStr );
    return stat;
}

MStatus uninitializePlugin( MObject obj)
{
    MStatus stat;
    MString errStr;
    MFnPlugin plugin( obj );
```

```
    stat = plugin.deregisterCommand( "antisimetria" );
    if ( !stat )
    {
            errStr = "deregisterCommand failed";
            goto error;
    }

    stat = plugin.deregisterNode( AntiSimetriaNode::id );
    if( !stat )
    {
            errStr = "deregisterNode failed";
            goto error;
    }

    return stat;

error:

    stat.perror( errStr );
    return stat;
}
```

AntiSimetriaNode.cpp

```
//
// Copyright (C) 2008 Fernando Garcia Torcelly
//

#include <fstream.h>
#include <istream.h>
#include "AntiSimetriaNode.h"
#include <maya/MPlug.h>
#include <maya/MTime.h>
#include <maya/MDataBlock.h>
#include <maya/MDataHandle.h>
#include <maya/MArrayDataHandle.h>
#include <maya/MGlobal.h>
#include <maya/MFloatPointArray.h>
#include <maya/MFnUnitAttribute.h>
#include <maya/MFnTypedAttribute.h>
#include <maya/MFnGenericAttribute.h>
#include <maya/MFnMeshData.h>
#include <maya/MFnMesh.h>
#include <maya/MPointArray.h>
#include <maya/MAngle.h>
#include <assert.h>
```

```
#include <float.h>
#include <maya/MArgList.h>

const MTypeId AntiSimetriaNode::id( 0x00338 );

MObject AntiSimetriaNode::inputSurfaceOrig;
MObject AntiSimetriaNode::inputSurface;
MObject AntiSimetriaNode::outputSurfaceantisim;
MObject AntiSimetriaNode::dirX;
MObject AntiSimetriaNode::dirY;
MObject AntiSimetriaNode::dirZ;
MObject AntiSimetriaNode::dirMX;
MObject AntiSimetriaNode::dirMY;
MObject AntiSimetriaNode::dirMZ;


MStatus AntiSimetriaNode::compute( const MPlug& plug, MDataBlock&
data )
{
    MStatus stat;

    if( plug == outputSurfaceantisim )
    {
            MDataHandle inputSurfaceOrigHnd =
            data.inputValue( inputSurfaceOrig );
            MDataHandle inputSurfaceHnd =
            data.inputValue( inputSurface );
            MDataHandle outputSurfaceantisimHnd =
            data.outputValue( outputSurfaceantisim );
            MDataHandle dirXHnd = data.outputValue( dirX );
            MDataHandle dirYHnd = data.outputValue( dirY );
            MDataHandle dirZHnd = data.outputValue( dirZ );
            MDataHandle dirMXHnd = data.outputValue( dirMX );
            MDataHandle dirMYHnd = data.outputValue( dirMY );
            MDataHandle dirMZHnd = data.outputValue( dirMZ );


            // Activar superficie Original
            MObject inputSurfaceOrigObj =
            inputSurfaceOrigHnd.asMesh();

            MFnMesh surfaceOrigFn;
            MFnMeshData surfaceDataOrigFn;
            MObject newSurfaceDataOrig = surfaceDataOrigFn.create();

            // Copia en el MObject newSurfaceData la malla original
            // sin mover
```

```
surfaceOrigFn.copy( inputSurfaceOrigObj,
newSurfaceDataOrig );

// SurfaceFn controla la malla original copiada
surfaceOrigFn.setObject( newSurfaceDataOrig );


// Se obtiene los puntos de la parte original
MPointArray ArrayOrigptos;
surfaceOrigFn.getPoints(ArrayOrigptos, MSpace::kWorld);
unsigned int numPtosIni=surfaceOrigFn.numVertices();
unsigned int numPoly=surfaceOrigFn.numPolygons();

// Inicializa valores que se necesitan
unsigned int index,i;
double orig[numPtosIni][4];
double resultado;
ArrayOrigptos.get(orig);


// Copia en el inputSurfaceObj la malla final
MObject inputSurfaceObj = inputSurfaceHnd.asMesh();

// Pone en MObject newSurfaceData datos nuevo y luego
// copia en ellos la malla deformada.
// Finalmente pone en surfaceFinFn esta malla deformada.
MFnMeshData surfaceDataFinFn;
MFnMesh surfaceFinFn;
MObject newSurfaceDataFin = surfaceDataFinFn.create();
surfaceFinFn.copy( inputSurfaceObj, newSurfaceDataFin );
surfaceFinFn.setObject( newSurfaceDataFin );


// Aumenta arrayenteros al doble del numero de caras
MIntArray arrayenteros;
for(index=0 ; index< numPoly ; index++) {
      arrayenteros.append(index);
}

// Al surfaceFn (malla deformada) se le añade el doble de
// caras
surfaceFinFn.duplicateFaces(arrayenteros, NULL);

// Obtiene numero de vertices del mallado completo
// (duplicado)
unsigned int numPtos=surfaceFinFn.numVertices();
```

```
// Se obtiene los puntos de la parte original
MPointArray Arrayptosdup;
surfaceFinFn.getPoints(Arrayptosdup, MSpace::kWorld);

// Arrayptosdup y Dest es matriz con valores de todos los
// puntos deformados (duplicados)
// La mitad es la original y la otra mitad es original
// esperando ser modificado
double dest[numPtos][4];
Arrayptosdup.get(dest);

// Bucle donde se modifican los puntos segun direccion de
// antisimetria
if(dirXHnd.asBool()) {
      // Inicializa el Xmaspos
      double Xmaspos;
      MPoint PtoOrig = ArrayOrigptos[0];
      Xmaspos = PtoOrig[0];

      // Mira que el Xmaspos sea mas positivo en X que el
      // punto siguiente
      for(i=1 ; i < numPtosIni ; i++) {
      MPoint PtoOrig = ArrayOrigptos[i];
            if(Xmaspos - PtoOrig[0] < 0) {
                  Xmaspos = PtoOrig[0];
            }
      }

      for(index=0 ; index <= numPtosIni ; index++)
      {
            // Pone los puntos duplicados de la malla
            // original en forma antisimetrica
            resultado = Xmaspos + (Xmaspos - orig[index]
            [0]) + ( dest[index][0] - orig[index][0]);
            Arrayptosdup.set(index, resultado, dest[index]
            [1], dest[index][2], 1);
      }
}

       if(dirYHnd.asBool()) {
             // Inicializa el Ymaspos
             double Ymaspos;
             MPoint PtoOrig = ArrayOrigptos[1];
             Ymaspos = PtoOrig[1];

            // Mira que el Xmaspos sea mas positivo en X
            // que el punto siguiente
             for(i=1 ; i < numPtosIni ; i++) {
```

```
                MPoint PtoOrig = ArrayOrigptos[i];
                if(Ymaspos - PtoOrig[1] < 0) {
                      Ymaspos = PtoOrig[1];
                }
          }

          for(index=0 ; index <= numPtosIni ; index++)
          {
          // Pone los puntos duplicados de la malla
          // original en forma antisimetrica
                resultado = Ymaspos + (Ymaspos -
                orig[index][1]) + ( dest[index][1] -
                orig[index][1]);
                Arrayptosdup.set(index, dest[index][0],
                resultado, dest[index][2], 1);
          }
    }

     if(dirZHnd.asBool()) {
          // Inicializa el Ymaspos
          double Zmaspos;
          MPoint PtoOrig = ArrayOrigptos[2];
          Zmaspos = PtoOrig[2];

          // Mira que el Xmaspos sea mas positivo en X
          // que el punto siguiente
          for(i=1 ; i < numPtosIni ; i++) {
                MPoint PtoOrig = ArrayOrigptos[i];
                if(Zmaspos - PtoOrig[2] < 0) {
                      Zmaspos = PtoOrig[2];
                }
          }

          for(index=0 ; index <= numPtosIni ; index++)
          {
          // Pone los puntos duplicados de la malla
          // original en forma antisimetrica
          resultado = Zmaspos + (Zmaspos - orig[index]
          [2]) + ( dest[index][2] - orig[index][2]);
          Arrayptosdup.set(index, dest[index][0],
          dest[index][1], resultado, 1);
          }
     }

     if(dirMXHnd.asBool()) {
          // Inicializa el Xmaspos
          double Xmasneg;
          MPoint PtoOrig = ArrayOrigptos[0];
```

```
        Xmasneg = PtoOrig[0];

      // Mira que el Xmaspos sea mas positivo en X
      // que el punto siguiente
      for(i=1 ; i < numPtosIni ; i++) {
            MPoint PtoOrig = ArrayOrigptos[i];
            if(Xmasneg - PtoOrig[0] > 0) {
                  Xmasneg = PtoOrig[0];
            }
       }

       for(index=0 ; index <= numPtosIni ; index++)
       {
      // Pone los puntos duplicados de la malla
      // original en forma antisimetrica
      resultado = Xmasneg + (Xmasneg - orig[index]
      [0]) + ( dest[index][0] - orig[index][0]);
      Arrayptosdup.set(index, resultado, dest[index]
      [1], dest[index][2], 1);
       }
}

if(dirMYHnd.asBool()) {

       // Inicializa el Ymaspos
       double Ymasneg;
       MPoint PtoOrig = ArrayOrigptos[1];
       Ymasneg = PtoOrig[1];

      // Mira que el Xmaspos sea mas positivo en X
      // que el punto siguiente
      for(i=1 ; i < numPtosIni ; i++) {
            MPoint PtoOrig = ArrayOrigptos[i];
            if(Ymasneg - PtoOrig[1] > 0) {
                  Ymasneg = PtoOrig[1];
            }
      }

      for(index=0 ; index <= numPtosIni ; index++)
      {
      // Pone los puntos duplicados de la malla
      // original en forma antisimetrica
      resultado = Ymasneg + (Ymasneg - orig[index]
      [1]) + ( dest[index][1] - orig[index][1]);
      Arrayptosdup.set(index, dest[index][0],
      resultado, dest[index][2], 1);
      }
}
```

```
                    if(dirMZHnd.asBool()) {
                          // Inicializa el Ymaspos
                          double Zmasneg;
                          MPoint PtoOrig = ArrayOrigptos[2];
                          Zmasneg = PtoOrig[2];

                         // Mira que el Xmaspos sea mas positivo en X
                         // que el punto siguiente
                         for(i=1 ; i < numPtosIni ; i++) {
                                MPoint PtoOrig = ArrayOrigptos[i];
                                if(Zmasneg - PtoOrig[2] > 0) {
                                      Zmasneg = PtoOrig[2];
                                }
                         }

                         for(index=0 ; index <= numPtosIni ; index++)
                         {
                         // Pone los puntos duplicados de la malla
                         // original en forma antisimetrica
                         resultado = Zmasneg + (Zmasneg - orig[index]
                         [2]) + ( dest[index][2] - orig[index][2]);
                         Arrayptosdup.set(index, dest[index][0],
                         dest[index][1], resultado, 1);
                         }
                   }

              // Coloca la parte antisimetrica
              surfaceFinFn.setPoints(Arrayptosdup, MSpace::kObject);

              // Activar superficie
              surfaceFinFn.updateSurface();

              outputSurfaceantisimHnd.set( newSurfaceDataFin );

              data.setClean( plug );

    }

    return stat;
}

void *AntiSimetriaNode::creator()
{
    return new AntiSimetriaNode();
}

MStatus AntiSimetriaNode::initialize( )
```

```
{
    // Inicia parametro
    // Status
    MStatus stat;

    MFnTypedAttribute tAttr;
    MFnNumericAttribute nAttr;
    MFnGenericAttribute gAttr;


    inputSurface = tAttr.create( "inputSurface", "is",
    MFnMeshData::kMesh );

    inputSurfaceOrig = tAttr.create( "inputSurfaceOrig", "io",
    MFnMeshData::kMesh );

    outputSurfaceantisim = tAttr.create( "outputSurfaceantisim", "os",
    MFnMeshData::kMesh );

    dirX = nAttr.create("dirx","dx", MFnNumericData::kBoolean);

    dirMX = nAttr.create("dirmx","dmx", MFnNumericData::kBoolean);

    dirY = nAttr.create("diry","dy", MFnNumericData::kBoolean);

    dirMY = nAttr.create("dirmy","dmy", MFnNumericData::kBoolean);

    dirZ = nAttr.create("dirz","dz", MFnNumericData::kBoolean);

    dirMZ = nAttr.create("dirmz","dmz", MFnNumericData::kBoolean);


    addAttribute( inputSurface );
    addAttribute( inputSurfaceOrig );
    addAttribute( outputSurfaceantisim );
    addAttribute( dirX );
    addAttribute( dirY );
    addAttribute( dirZ );
    addAttribute( dirMX );
    addAttribute( dirMY );
    addAttribute( dirMZ );

    attributeAffects( inputSurface, outputSurfaceantisim );

    return MS::kSuccess;
}
```

18. Color PerVertex

pluginMain.cpp

```
//
// Copyright (C) 2008 Fernando Garcia Torcelly
//

#include "VtxColorsNode.h"
#include "VtxColorsCmd.h"
#include <maya/MFnPlugin.h>

MStatus initializePlugin( MObject obj )
{
    MStatus stat;
    MString errStr;
    MFnPlugin plugin( obj, "Fernando Garcia Torcelly", "1.0", "Any");

    stat = plugin.registerCommand( "vtxcolors",
VtxColorsCmd::creator );
    if ( !stat )
    {
            errStr = "registerCommand failed";
            goto error;
    }

    stat = plugin.registerNode( "vtxcolors", VtxColorsNode::id,
VtxColorsNode::creator, VtxColorsNode::initialize );
    if ( !stat )
    {
            errStr = "registerNode failed";
            goto error;
    }

    return stat;

error:

    stat.perror( errStr );
    return stat;
}

MStatus uninitializePlugin( MObject obj)
{
    MStatus stat;
    MString errStr;
    MFnPlugin plugin( obj );
```

```
    stat = plugin.deregisterCommand( "vtxcolors" );
    if ( !stat )
    {
            errStr = "deregisterCommand failed";
            goto error;
    }

    stat = plugin.deregisterNode( VtxColorsNode::id );
    if( !stat )
    {
            errStr = "deregisterNode failed";
            goto error;
    }

    return stat;

error:

    stat.perror( errStr );
    return stat;
}
```

VtxColorsNode.cpp

```
//
// Copyright (C) 2008 Fernando Garcia Torcelly
//

#include <fstream.h>
#include <istream.h>
#include "VtxColorsNode.h"
#include <maya/MPlug.h>
#include <maya/MTime.h>
#include <maya/MDataBlock.h>
#include <maya/MDataHandle.h>
#include <maya/MGlobal.h>
#include <maya/MFnUnitAttribute.h>
#include <maya/MFnTypedAttribute.h>
#include <maya/MFnMeshData.h>
#include <maya/MFnMesh.h>
#include <maya/MPointArray.h>
#include <maya/MAngle.h>
#include <assert.h>
#include <float.h>
#include <regex.h>

const MTypeId VtxColorsNode::id( 0x00334 );
```

```
MObject VtxColorsNode::inputSurface;
MObject VtxColorsNode::outputSurface;
MObject VtxColorsNode::time;
MObject VtxColorsNode::namefile;
MObject VtxColorsNode::amplitud;
MObject VtxColorsNode::velocidad;
MObject VtxColorsNode::maximo;
MObject VtxColorsNode::Boolmaximo;

MStatus VtxColorsNode::compute( const MPlug& plug, MDataBlock& data )
{
MStatus stat;

if( plug == outputSurface )
    {
    MDataHandle inputSurfaceHnd = data.inputValue( inputSurface );
    MDataHandle outputSurfaceHnd = data.outputValue( outputSurface );
    MDataHandle namefileHnd = data.outputValue( namefile );
    MDataHandle timeHnd = data.inputValue( time );
    MDataHandle amplitudHnd = data.outputValue( amplitud );
    MDataHandle velocidadHnd = data.outputValue( velocidad );
    MDataHandle maximoHnd = data.outputValue( maximo );
    MDataHandle boolmaximoHnd = data.outputValue( Boolmaximo );

    // El MObject toma la malla importada
    MObject inputSurfaceObj = inputSurfaceHnd.asMesh();

    // NewSurfaceData toma el nuevo MFnMeshData
    MFnMeshData surfaceDataFn;
    MObject newSurfaceData = surfaceDataFn.create();

    // El surfaceFn toma newSurfaceData=inputSurfaceObj (malla de
    // entrada)
    MFnMesh surfaceFn;
    surfaceFn.copy( inputSurfaceObj, newSurfaceData );
    surfaceFn.setObject( newSurfaceData );

            // Abre el fichero indicado
            MString myfile=namefileHnd.asString();
            double Amplitud=amplitudHnd.asDouble();
            double Velocidad=velocidadHnd.asDouble();
            double Maximo=maximoHnd.asDouble();
            bool boolmaximo=boolmaximoHnd.asBool();
            MTime tframe=timeHnd.asTime();
            double frame=tframe.as(MTime::kFilm);
```

```
// Declaracion de variables para lectura de fichero de
// movimiento
float frec;
char Puntos[250];
char *PtoSep;


// Abre fichero de datos
fstream filein;
filein.open(myfile.asChar(),ios::in);

// Si no se puede abrir el fichero
if(!filein) {
      MGlobal::displayError(MString("No se ha podido abrir
      el fichero"));
      return MS::kFailure;
}


// Lee el tipo de fichero
char NumLinPun[100];
filein.getline(NumLinPun,99,'\n');
PtoSep=strtok(NumLinPun," |\t");


// Inicialización de variables para regex
regex_t retmp;
regmatch_t mtmp;
int res;


// Guarda el tipo del fichero
bool temporal = false;
bool frecuencia = false;
res = regcomp(&retmp, "^temporal$|^temporal[\|]{1}$",
REG_EXTENDED);
res = regexec(&retmp, PtoSep, (size_t) 1, &mtmp, 0);
if (res!=0) {
      frecuencia=true;
}
else {
      temporal=true;
}
if(frecuencia&&temporal){
      MGlobal::displayError(MString("Error recogiendo tipo
      de fichero"));
      return MS::kFailure;
}
```

```
if(frecuencia) {

// Lee la frecuencia que tiene el fichero
filein.getline(NumLinPun,99,'\n');
PtoSep=strtok(NumLinPun," |\t");


// Hace un cast de char a float
frec=atof(PtoSep);


// Numeros de puntos a leer
unsigned int NumPun;
filein.getline(NumLinPun,99,'\n');
PtoSep=strtok(NumLinPun," |\t");


// Hace cast de char a int
NumPun=atoi(PtoSep);


// Inicializa variables necesarias para recoleccion de
// datos y la matriz
float PunMov[NumPun][10];
int i,j;

// Comienza a leer
for(i=0,j=0;i<=(NumPun-1);i++)
{
      filein.getline(Puntos,199,'\n');
      PtoSep=strtok(Puntos," |\t");

      PunMov[i][j]=atof(PtoSep);
      j++;

      PtoSep=strtok(NULL," |\t");

      while(PtoSep!=NULL)
      {
            PunMov[i][j]=atof(PtoSep);
            PtoSep=strtok(NULL," |\t");
            j++;
      }

      j=0;
}
```

```
filein.close();

//  ----------------------------------------


// Inicializa variables para chequeos
float SegFrame=0.04166666*float(Velocidad);
float varX,varY,varZ;
float c1, c2, c3;
float Test,temp;

if(!boolmaximo) {
      Maximo=0;

// Chequea Maximos
      for(i=0;i<NumPun;i++)
      {
            varX=Amplitud*(sqrt((PunMov[i][4]*PunMov[i][4])
            +(PunMov[i][5]*PunMov[i][5])));
            varY=Amplitud*(sqrt((PunMov[i][6]*PunMov[i][6])
            +(PunMov[i][7]*PunMov[i][7])));
            varZ=Amplitud*(sqrt((PunMov[i][8]*PunMov[i][8])
            +(PunMov[i][9]*PunMov[i][9])));

            // Mide desplazamiento
            Test=sqrt((varX*varX)+(varY*varY)+(varZ*varZ));
            if(Test>Maximo)
            {
                  Maximo=Test;
            }
      }
}


// Chequea Colores
for(i=0;i<=NumPun;i++)
{
      varX=Amplitud*(sqrt((PunMov[i][4]*PunMov[i][4])+
      (PunMov[i][5]*PunMov[i][5])))*(
      cos(frec*(SegFrame*frame)));
      varY=Amplitud*(sqrt((PunMov[i][6]*PunMov[i][6])+
      (PunMov[i][7]*PunMov[i][7])))*(
      cos(frec*(SegFrame*frame)));
      varZ=Amplitud*(sqrt((PunMov[i][8]*PunMov[i][8])+
      (PunMov[i][9]*PunMov[i][9])))*(
      cos(frec*(SegFrame*frame)));
```

```
                    // Mide desplazamiento
                    if(Maximo!=0)
                          Test=(sqrt((varX*varX)+(varY*varY)+
                          (varZ*varZ)))/Maximo;
                    else
                          Test=0;

                    // Estableciendo los parametros de color
                    if(Test>0.75 && Test<=1)
                    {
                          c2=1;
                          c3=0.9;
                    }
                    else if(Test>0.5 && Test<=0.75)
                    {
                          c2=0.8;
                          c3=0.8;
                    }
                    else if(Test>0.25 && Test<=0.5)
                    {
                          c2=1;
                          c3=0.75;
                    }
                    else
                    {
                          c2=0.8;
                          c3=0.945;
                    }

                    // Temporal
                    temp=Test*240;
                    c1=240-temp;

                    MColor hsvValues(MColor::kHSV, c1, c2, c3);
                    stat=surfaceFn.setVertexColor(hsvValues,i,NULL);

              }

              }

    surfaceFn.updateSurface();

    outputSurfaceHnd.set( newSurfaceData );

    data.setClean( plug );
    }
else
    stat  = MS::kUnknownParameter;
```

```
return stat;
}

void *VtxColorsNode::creator()
{
return new VtxColorsNode();
}

MStatus VtxColorsNode::initialize()
{
    MFnTypedAttribute tAttr;
    MFnUnitAttribute uAttr;
    MFnNumericAttribute nAttr;

    time = uAttr.create("time", "t", MFnUnitAttribute::kTime);

    inputSurface = tAttr.create( "inputSurface", "is",
MFnMeshData::kMesh  );

    outputSurface = tAttr.create( "outputSurface", "os",
MFnMeshData::kMesh  );

    amplitud = nAttr.create( "amplitud", "am",
MFnNumericData::kDouble  );

    velocidad = nAttr.create( "velocidad", "ve",
MFnNumericData::kDouble  );

    maximo = nAttr.create( "maximo", "ma", MFnNumericData::kDouble  );

    Boolmaximo = nAttr.create( "Boolmaximo", "bm",
MFnNumericData::kBoolean  );

    namefile = tAttr.create( "namefile", "nf", MFnData::kString  );

    addAttribute( time );
    addAttribute( inputSurface );
    addAttribute( outputSurface );
    addAttribute( namefile );
    addAttribute( amplitud );
    addAttribute( velocidad );
    addAttribute( maximo );
    addAttribute( Boolmaximo );

    attributeAffects( time, outputSurface );
    attributeAffects( maximo, outputSurface );
    attributeAffects( inputSurface, outputSurface );
```

```
    return MS::kSuccess;
}
```

BIBLIOGRAFÍA

Recursos bibliográficos.

- Manual imprescindible de C/C++. Miguel Ángel Acera García. Anaya Multimedia. Madrid. 2005.

- La Biblia de Visual C++. Net. Tom Archer, Andrew Whitechapel. Anaya Multimedia. Madrid. 2003.

- Ideas generales sobre el Método de los Elementos de Contorno. Francisco Beltrán. E.T.S. Ingenieros Industriales de Madrid. 1999.

- C/C++ : Curso de Programación. Francisco Javier Ceballos Sierra. Ra-Ma. Paracuellos del Jarama. 2007.

- A Comparative Study of Acceleration Techniques for Geometric Visualization. Pascual Castelló, José Francisco Ramos, Miguel Chover. Departamento de Lenguajes de Computación y Sistemas. Universidad Jaume I. Castellón de la Plana. 2004.

- Informática Gráfica. Juan Manuél Corchado Rodriguez. Segundo ciclo Ingeniería Informática. Universidad de Salamanca. Salamanca. 2004.

- Maya 7. Marc André Guindon. Anaya multimedia. Madrid. 2006.

- Complete Maya Programming. Volume I. David Gould. Ed. Morgan Kaufmann. San Francisco, EE.UU. 2003.

- Complete Maya Programming. Volume II. David Gould. Ed. Morgan Kaufmann. San Francisco, EE.UU. 2005.

- Cómo programar en C/C++ y Java. Harvey M. Deitel, Paul J. Deitel. Prentice-Hall Hispanoamericana. México. 2004.

- Los secretos de Maya. John Kundert Gibbs. Anaya Multimedia. Madrid. 2003.

- Maya 7. Curso Avanzado. Josep Molero. Ed. Inforbook's S.L. Barcelona. 2007.

- Maya 8/8.5. Josep Molero. Ed. Inforbook's S.L. Barcelona. 2007.

Recursos en línea.

- Autodesk Forum.
  http://discussion.autodesk.com

- CGSociety.
  http://forums.cgsociety.org

- C++ References.
  http://www.cplusplus.com

- Developer Apple.
  http://developer.apple.com

- EpicGames
  http://udn.epicgames.com/Two/UnrealVertexAnimation.html#When%20to%20use%20Vertex%20Animations?

- Finite Element Analisys.
  http://www.finiteelement.com/feawhite4.html

- Maya 2008 Help.
  http://download.autodesk.com/us/maya/2008help/wwhelp/wwhimpl/js/html/wwhelp.htm

- Mental Ray.
  http://www.mentalimages.com

- Renderosity.
  http://www.renderosity.com/news.php?viewStory=13361

- Rob The Block.
  http://www.robthebloke.org/mel/DATA_mesh.html

- USC Siggraph.
  http://imagine-it.org/uscsiggraph/index.php?option=com_content
  &task=view&id=85&Itemid=2