DEPARTAMENTO DE INFORMÁTICA

ACTORES SINTÉTICOS EN TIEMPO REAL: NUEVAS ESTRUCTURAS DE DATOS Y MÉTODOS PARA SU INTEGRACIÓN EN APLICACIONES DE SIMULACIÓN.

RAFAEL RODRÍGUEZ GARCÍA

UNIVERSITAT DE VALENCIA
Servei de Publicacions
2004

Aquesta Tesi Doctoral va ser presentada a Valencia el día 23 de Juliol de 2004 davant un tribunal format per:

- D. Roberto Vivó Hernando
- D. Alfonso Brazalez Guerra
- D. Alfredo Pina Calari
- D. Francisco V. Perales López
- D. Mariano Pérez Martines

Va ser dirigida per:
D. Francisco José Serún Arbeloa
D. Marcos Fernández Marín

Depòsit legal:
I.S.B.N.:84-370-0426-8
Edita: Universitat de València
Servei de Publicacions
C/ Artes Gráficas, 13 bajo
46010 València
Spain
Telèfon: 963864115

Departament d´Informàtica.

ACTORES SINTÉTICOS EN TIEMPO REAL: NUEVAS ESTRUCTURAS DE DATOS Y MÉTODOS PARA SU INTEGRACIÓN EN APLICACIONES DE SIMULACIÓN

TESIS DOCTORAL

RAFAEL RODRÍGUEZ GARCÍA

DIRECTORES:

DR. FRANCISCO JOSÉ SERÓN ARBELOA

DR. MARCOS FERNÁNDEZ MARÍN

Valencia, 2004

Universitat de València

Departament D'Informática.
Av Vicente Andrés Estellés,
46100 Burjassot (Valencia).

D. **Francisco Jose Serón Arbeloa** Catedrático de Universidad en el área de Lenguajes y Sistemas Informáticos de la Universidad de Zaragoza, y D. **Marcos Fernández Marín** profesor Titular de Universidad en el área de Ciencias de la Computación e Inteligencia Artificial de la Universitat de València,

HACEN CONSTAR:

que la presente tesis doctoral titulada "Actores Sintéticos en Tiempo Real: Nuevas Estructuras de Datos y Métodos para su Integración en Aplicaciones de Simulación", ha sido realizada bajo su dirección por D. Rafael Rodríguez García, el cual la presenta para optar al grado de Doctor en Ingeniería Informática.

Y para que así conste, firman el presente certificado en Valencia a **1** de Abril del dos mil cuatro.

Fdo: Francisco Jose Serón Arbeloa Fdo: Marcos Fernández Marín

ACTORES SINTÉTICOS EN TIEMPO REAL:
NUEVAS ESTRUCTURAS DE DATOS Y MÉTODOS PARA SU INTEGRACIÓN EN APLICACIONES DE SIMULACIÓN

MEMORIA PRESENTADA PARA OPTAR AL GRADO DE DOCTOR EN INGENIERÍA INFORMÁTICA

POR:

D. RAFAEL RODRÍGUEZ GARCÍA

DIRIGIDA POR:

DR. D. FRANCISCO JOSÉ SERÓN ARBELOA.

DR. D. MARCOS FERNÁNDEZ MARÍN

DEPARTAMENTO DE INFORMÁTICA. UNIVERSIDAD DE VALENCIA.

ABRIL 2004

Agradecimientos.

Deseo expresar mi agradecimiento a mis directores de Tesis por su apoyo tanto científico como personal a lo largo de la realización de este trabajo.

También a todos los compañeros del Grupo de Informática Gráfica Avanzada de la Universidad de Valencia (ARTEC), con los que he compartido tantas horas de trabajo e ilusión.

A todos los miembros del Instituto de Robótica y del Departamento de Electrónica e Informática de la Universidad de Valencia, con los que he tenido el gusto de compartir muchos buenos ratos. Y también a mis actuales compañeros de Brainstorm Multimedia.

Dedicatoria.

A mi madre, Julia, por su amor y apoyo constante.

A mi hermano, Juanjo, por su humanidad y nobleza.

A mi amor, Cristina, por compartir su vida conmigo.

A mi padre, Silverio, que ya no esta aquí, por su recuerdo y sus genes.

Marco de desarrollo del trabajo.

Si bien, ya existía un interés previo por parte del autor por la temática relacionada con los actores virtuales (cortometraje de animación "Bob"[1]), este trabajo de tesis comenzó a gestarse, en el marco del proyecto Europeo ARTIST[2], en el cual el autor actuó como desarrollador de estructuras de bajo nivel para el soporte de actores virtuales, así como de integrador de los desarrollos en esta área realizados por diferentes partners del proyecto: modelos geométricos (*Art &Magic*), módulos de keyframing (*APD*), cinemática inversa (*GIGA*), y modelo comportamental (*Norks Regnesentral*). La financiación del proyecto ARTIST finalizo en Febrero de 1998, desde este momento, el autor continuo con su actividad investigadora en actores virtuales de una forma personal, compatibilizando el desarrollo de la tesis con la docencia en la Facultad de Ingeniería Informática de la Universidad de Valencia, y el trabajo como ingeniero de I+D en la empresa Brainstorm Multimedia, en actividades estrechamente relacionadas con el desarrollo de grafos de escena multiplataforma. La mayor parte de este trabajo ha sido realizado en este último periodo de tiempo, manteniendo el contacto con la Universidad mediante encuentros periódicos con sus directores de tesis.

Las publicaciones presentadas hasta el momento actual en relación con los contenidos de este trabajo han sido los siguientes:

"Adding Support for High-Level Skeletal Animation". F.J. Serón, R.Rodríguez, E. Cerezo, A. Pina. IEEE Transactions on Visualization and Computer Graphics. pp: 360-372. Octubre 2002.

"Integrating Synthetic Actors In Simulation Applications". R. Rodríguez. M. E. Martínez, I. Coma, F.Martinez, F.J. Serón.. International Training & Education Conference ITEC´'99. The Hague. Nederlands. 1999.

"ArtGraph: Un entorno integrado de desarrollo y ejecución de aplicaciones 3D Tiempo Real". I. Coma, R. Rodríguez, M. Fernández, E. Martínez, P. Caselles. Proc. Congreso Español de Informática Gráfica.CEIG´98. pp. 81-94. Junio 1998.

---

[1] Cortometraje de Animación "Bob" 3.45 minutos. Desarrollado en el Dpto. de Narración Figurativa de la Facultad de BBAA de la UPV. Presentado en el Festival "Cinema Jove" de 1997

[2] ARTIST: Animation Package for Real-Time Simulation, ESPRIT Project E20102. Developed by LISITT (University of Valencia, Spain), GIGA(University of Zaragoza, Spain), Norks Regnesentral (Norway), APD (Spain) and ART&Magic (Belgium) 1997.

Índice General.

PARTE I. INTRODUCCIÓN.

PARTE II. ESTADO DEL ARTE.

PARTE III. ESTRUCTURAS Y MÉTODOS.

PARTE IV. IMPLEMENTACIÓN.

PARTE I

INTRODUCCIÓN

Capítulo 1. Introducción.

1.1 Índice.

1.2 Motivaciones.

Uno de los temas más investigados en los últimos años el mundo de la informática gráfica ha sido la animación de personajes articulados. Es un campo enormemente amplio en el que se involucran áreas tan variadas como la cinemática, la dinámica o la inteligencia artificial, y que tiene un extenso ámbito de aplicación práctica.

En el mundo de los gráficos por ordenador, existen dos líneas investigación que se encuentran en estados muy avanzados, pero han seguido caminos divergentes: de un lado la *Animación 3D* que ha centrado toda su atención en la calidad visual de las imágenes, y, de otro, la *Realidad Virtual* o *Informática Gráfica en Tiempo Real*, que ha intentado conseguir la mayor calidad visual posible, pero dando prioridad a la rapidez en la generación de las imágenes.

La gran mayoría de trabajos sobre animación de actores virtuales han sido realizados en el mundo de la *Animación 3D,* y sus esfuerzos centrados en conseguir un gran realismo visual, normalmente a costa del empleo de grandes máquinas y tiempos de cálculo elevados. Los resultados conseguidos en este campo han sido realmente espectaculares, sin embargo, este tipo de actores sintéticos no son adecuados para ser integrados en una aplicación de simulación en tiempo real, y es necesario enfocar el problema desde una óptica distinta.

La necesidad de actores virtuales dentro de los entornos de simulación es evidente: En un simulador de conducción son necesarias calles y vehículos, pero también peatones e incluso instructores; las reconstrucciones históricas necesitan guías virtuales; los entornos arquitectónicos necesitan personajes que los habiten etc. El empleo de actores virtuales en tiempo real es de gran utilidad en aplicaciones de ergonomía, psicología, robótica, medicina, multimedia, cine, simulación civil o videojuegos. El hecho de que el coste económico de la tecnología de realidad virtual se haya reducido notablemente en los últimos años, posibilita que su ámbito de aplicación real sea aún más amplio.

Sin embargo, los entornos virtuales siguen despoblados. La representación geométrica de un actor virtual de una calidad aceptable tiene un elevado coste computacional, y el elevado número de variables a manejar en la gestión de sus articulaciones eleva en gran medida la complejidad de la aplicación. Para los programas de animación 3D orientados a la generación de secuencias para el cine, éste es un problema relativo, puesto que no tienen especial problema en dedicar varias horas a definir y renderizar los movimientos de un actor en una escena determinada. Una simulación en tiempo real requiere que la escena entera sea dibujada varias veces por segundo, y esto fuerza a emplear estructuras y métodos de gestión especiales.

La forma tradicional de definición de una escena de simulación consiste en el empleo de un *Grafo de Escena* (*Scene Graph* en literatura inglesa), el cual es una estructura en forma de grafo, formada por

diversos tipos de nodos que ejercen distintos tipos de influencia sobre sus nodos hijos, bien sea estableciendo una agrupación, introduciendo algún tipo de poda o aplicando alguna transformación afín. Los nodos hijos de estas estructuras suelen ser los objetos geométricos a representar.

Los grafos de escena actuales están principalmente orientados a la representación de entornos estáticos, o con objetos que presentan movimientos simples, como por ejemplo aviones o coches, y presentan serias carencias a la hora de representar elementos con una estructura articulada compleja, o elementos cuya gestión de comportamiento tenga un coste computacional elevado.

En la actualidad existen varias líneas de investigación relacionadas con los actores virtuales en tiempo real, pero es norma común que traten los aspectos de más alto nivel, relacionados con la gestión de su comportamiento. También existen distintos tipos de aplicaciones que recurren a la utilización de este tipo de actores, pero suelen ser desarrollos totalmente verticales destinados a cubrir necesidades muy concretas del mercado.

Es necesario analizar las carencias de los grafos de escena actuales en la representación de actores virtuales, y proponer soluciones que permitan definir de una forma estándar, cualquier tipo de actor virtual, bien sea un humanoide, o cualquier otro tipo de estructura articulada compleja tanto de origen natural como artificial. Estas soluciones pasan por la definición de nuevos tipos de nodos, la búsqueda de nuevos métodos de gestión del grafo de escena, y también, la creación de nuevas estructuras de datos capaces de encapsular la gestión de su comportamiento, y otras características propias de los actores virtuales, que no habían sido planteadas hasta ahora por los grafos de escena tradicionales.

1.3 Objetivos.

El objetivo final de este trabajo, es conseguir un método estándar para la definición e integración de un conjunto elevado de actores virtuales en una aplicación de simulación. En la actualidad es posible definir un actor virtual sobre la base de las estructuras proporcionadas por un grafo de escena tradicional, sin embargo, su orientación a la representación de escenas estáticas hace que presente serias deficiencias.

La utilización de grafos de escena en el desarrollo de aplicaciones de simulación es muy antigua, y los métodos empleados son estables y bien conocidos. Debido a esto, en este trabajo no se pretende la definición de un nuevo tipo de grafo de escena, en su lugar, se realizará un análisis sobre el origen de las limitaciones existentes en los grafos de escena actuales a la hora de representar actores virtuales, y proponen ampliaciones que permitan que las aplicaciones de simulación puedan seguir funcionando de una forma similar a la actual, pero que, además, puedan contener y gestionar de un modo eficiente, un conjunto elevado de actores virtuales. Los grafos de escena actuales proporcionan diversas características que son útiles a la hora de la definición de un actor virtual:

- Un método para describir una estructura jerárquica que puede ser empleado en la definición de la estructura articulada del actor.
- Nodos de tipo transformación afín, a partir de los cuales se pueden definir las orientaciones de los puntos de articulación.
- Un método de CULL genérico que evita el dibujado de los objetos que se encuentran fuera de la pirámide de visión.
- Un método de gestión de Nivel de Detalle genérico, que permite hacer que los objetos que se encuentran lejos sean representados con un menor número de polígonos.
- Varios métodos para definir la geometría y la apariencia física de los objetos (propiedades de color, textura, respuesta a la luz, transparencia etc.).

El problema principal de la definición de actores virtuales a partir de un grafo de escena tradicional, es que los nodos de transformación afín no son suficientemente específicos y eficientes, y el hecho de que los actores apliquen gran número de transformaciones hace que los métodos de gestión de culling y nivel de detalle tradicionales resulten inadecuados. Adicionalmente, existen aspectos que no han sido nunca contemplados por los grafos de escena tradicionales, y que presentan una importancia vital en el caso de los actores virtuales: Necesidad de simplificar el control (elevado número de parámetros), de mejora de la organización (elevado número de nodos necesarios para definir un actor virtual), necesidad de llevar una correcta gestión del comportamiento, necesidad de separar el aspecto geométrico de las informaciones de alto nivel, de ser integrados en simuladores que manejen información macroscópica, o de hacer que sea capaz de integrar módulos de ampliación.

Los principales objetivos de este trabajo podrían ser concretados en la siguiente serie de puntos:

- El análisis y la solución del cuello de botella existente en los grafos de escena tradicionales en relación con la aplicación de transformaciones. Es necesario realizar una *gestión óptima de todas las operaciones con matrices de transformación*, puesto que son operaciones muy costosas, que se emplean de forma exhaustiva durante el procesado de los actores virtuales.
- La definición de *nuevos tipos de nodos* que actúen como un elemento modular estándar a partir del cual sea posible ensamblar actores virtuales eficientes, y que además, puedan relacionarse con los nodos tradicionales. Los nodos de transformación afín actuales son demasiado genéricos y poco eficientes, no resultando adecuados para definir la estructura articulada de un actor virtual.
- La búsqueda de un *método que permita simplificar el control de los actores* existentes en el grafo de escena, (un actor virtual está formado por muchos nodos distintos, y su control se realiza a partir de un conjunto elevado de parámetros). Es necesario proporcionar un método de encapsulación de la información del actor virtual, de tal modo que el manejo por parte del usuario resulte sencillo.
- La creación de una estructura que permita separar la información geométrica del actor, de su información lógica. Los nodos del grafo de escena han de contener información suficiente para definir el modelo geométrico del actor, pero los actores son entidades complejas, formados, además, por un modelo motor y un modelo comportamental. No tiene sentido que estos modelos sean almacenados en los nodos del grafo de escena, además este tipo de informaciones es común a todos los actores de una misma especie (así por ejemplo el método de alto nivel para caminar es común a todos los actores de tipo humanoide existentes en una simulación). Se definirá una estructura, independiente del grafo de escena, que almacene las características comunes y de alto nivel de una determinada "especie" de actor.
- Permitir que las capacidades de los actores virtuales sean ampliables. La investigación con respecto a actores virtuales está aún en su fase inicial, es básico que un sistema que permita incorporar actores virtuales en una aplicación de simulación preste especial atención a su extensibilidad, preocupándose de proporcionar un substrato básico suficientemente robusto y estándar para poder facilitar la realización de este tipo de ampliaciones.
- Capacidad de simulación macroscópica. El hecho de que una simulación pueda requerir un número muy elevado de actores (varios cientos de miles), hace necesario que su gestión pueda ser llevada a cabo mediante un conjunto de parámetros de tipo macroscópico, y que sea posible la creación y eliminación de actores en tiempo de ejecución.
- La definición de un *método de culling especial* que minimice el número de operaciones con matrices de transformación, así como las operaciones de recálculo e intersección con las *bounding spheres* de los actores virtuales, y que, además, actúe sobre la gestión del comportamiento.
- La definición de un *método de gestión de Nivel de Detalle especial* que minimice el número de operaciones con matrices de transformación, los costes del cálculo de distancias, y que, además, sea capaz de actuar, no solamente sobre la geometría, sino también sobre la topología, y sobre el coste derivado de la gestión del comportamiento.
- La descripción de un método, adaptado a las características de una simulación con actores virtuales, que indique como se debería de distribuir el trabajo entre las distintas CPUs en el caso de disponer de un sistema multiprocesador.

1.4 Organización.

Este documento se encuentra estructurado en 5 partes:

En la **Parte I** se realiza un estudio sobre el ***Estado del Arte*** de la Informática Gráfica en Tiempo Real y también de los Actores Sintéticos en Tiempo Real. Respecto a la Informática Gráfica en Tiempo Real (capítulo 2), se describirán las características básicas de las librerías de bajo nivel empleadas para acceder al hardware gráfico, así como de las librerías de alto nivel empleadas en las aplicaciones de simulación, y que están basadas en la utilización de grafos de escena. Se mostrarán las características comunes a todos los grafos de escena, prestando especial atención a los nodos básicos, la gestión de culling y nivel de detalle y también al multiproceso, y también las particularidades de los grafos de escena actuales más representativos. Respecto a los Actores Sintéticos en Tiempo Real (capítulo 3), se presentarán las características y estado del arte del modelo multicapa (comportamental, motora, geométrica), empleado en las investigaciones sobre actores virtuales. También se mostrarán los distintos campos de aplicación existentes y se prestará especial atención a los aspectos relacionados con la simulación en tiempo real, los modelos de representación de su geometría, y a las distintas propuestas de estandarización surgidas hasta el momento.

La **Parte II** contiene la descripción de todas las ***aportaciones*** realizadas en este trabajo. Está formado por cuatro capítulos. El primero de ellos (capítulo 4), analiza la forma de integración de los actores virtuales en el *Grafo de Escena*, proponiendo y justificando la creación de dos nuevos tipos de nodos (*Actor* y *Skeleton*), que actúan como elemento modular básico para el ensamblado de cualquier tipo de actor virtual. En el segundo capitulo (capítulo 5), se propone la existencia de un tipo de estructura independiente del grafo de escena que almacene las informaciones de alto nivel, y actúe como soporte para la realización de futuras ampliaciones, y para la simulación macroscópica. En el tercer capitulo (capítulo 6), se muestra la necesidad de realizar un procesado especial de las matrices de transformación, y se proponen métodos de *Culling* y gestión de *Nivel de Detal*le específicos, y también se realiza un análisis sobre la forma de gestión de actores virtuales en un sistema multiprocesador. El cuarto capítulo (capítulo 7), ofrece conclusiones sobre la mejora computacional introducida por la utilización de las distintas estructuras y métodos propuestos en este trabajo.

La **Parte III** presenta una ***implementación práctica*** de las estructuras y métodos propuestos en este trabajo (capitulo 8), consistente en un prototipo de librería con un interfaz de programación en ANSI C, la descripción de un método que permite hacer que la gestión de actores sea independiente del grafo de escena, y la propuesta de una arquitectura multi-capa que permite afrontar el problema de la creación de una aplicación con actores virtuales de una forma modular. En esta parte, también se muestra un ***ejemplo de aplicación*** (capítulo 9), en el cual se definen una serie de módulos que serán añadidos sobre una aplicación ya existente, de tal modo que incorpore una cantidad muy elevada de actores virtuales. En este caso se ampliará la aplicación de visualización de bases de datos de *OpenGl Performer* conocida como

"perfly", de tal modo que visualice un escenario consistente en una pequeña ciudad (*"Performer Town")* y, haciendo que sobre su cielo realicen distintos tipos de actividades varios miles de aves. Se podrá observar como su presencia no altera los requisitos de una aplicación en tiempo real, y como el empleo de las estructuras propuestas en este trabajo, facilita la definición de la escena, y la gestión de la aplicación.

Por último, La **Parte IV** ofrece ***conclusiones*** acerca de los resultados obtenidos, mostrando el grado de cumplimiento de los objetivos fijados inicialmente, exponiendo las futuras líneas de investigación abiertas a partir de este trabajo, así como las líneas de investigación ya existentes que resultan beneficiadas.

PARTE I

ESTADO DEL ARTE

Capítulo 2. Informática Gráfica en Tiempo Real: Estado del Arte.

2.1 Índice.

2.2 Introducción.

La representación una escena tridimensional en un ordenador requiere que toda la información referente a la descripción de los objetos, la iluminación, la atmósfera y los puntos de vista sean transformados en una imagen 2D que presente una calidad suficiente como para proporcionar una sensación realista.

La informática gráfica actual dispone de múltiples métodos para proporcionar definiciones matemáticas de los objetos tridimensionales, así, se pueden emplear definiciones de geometría sólida (CSG), en superficies paramétricas, campos potenciales, fractales, etc. De igual modo, existen múltiples métodos para definir las propiedades físicas de los objetos que constituyen la escena tridimensional, y de iluminar y transformar estas escenas en imágenes planas **[FOL91].** En el caso de quese esté buscando la obtención imágenes estáticas, o bien de secuencias grabadas de este tipo de imágenes, es posible recurrir a cualquiera de estos métodos. Normalmente estas tareas son llevadas a cabo en la CPU, y suelen presentar una complejidad computacional tan elevada que hace que el tiempo empleado en generar cada imagen sea habitualmente de varios minutos o incluso varias horas.

Frente a este tipo de aplicaciones en las que prima la calidad de la imagen (a costa del tiempo de cálculo que sea necesario), existen otro tipo de aplicaciones, en las que es necesario que la generación de imágenes se produzca con una rapidez tal que posibilite la interactividad (por encima de 8 imágenes/segundo), o la percepción contínua del movimiento (tasas de generación de imágenes en torno a los 50 imágenes/segundo). Con este objetivo, se han diseñado tarjetas gráficas especialmente orientadas a descargar a la CPU de las tareas relacionadas con el renderizado de las imágenes tridimensionales. No todos los métodos software empleados en la actualidad para conseguir imágenes 3D, pueden ser fácilmente implementadas dentro de una estructura hardware. Los métodos empleados para generar estas imágenes en tiempo real estén fuertemente condicionados por las características del hardware gráfico existente, existiendo una línea de investigación paralela a la **Informática Gráfica Tradicional**, cuyo objetivo es también conseguir la máxima calidad gráfica posible, pero siempre enmarcada por las limitaciones de este hardware gráfico. Esta línea de investigación paralela suele recibir el nombre genérico de **Informática Gráfica en Tiempo Real**.

Para poder realizar representaciones visuales de una escena tridimensional en tiempo real, se ha de disponer de un formato de datos, según el cual, los objetos y sus características gráficas estén definidos a partir primitivas optimizadas para trabajar con este tipo de hardware gráfico. El hardware gráfico dispone de uno o varios procesadores encargados de procesar dichas primitivas (transformar la posición espacial de los objetos y realizar su proyección, cálculos de iluminación, ocultación entre objetos, cálculos de sombreado, mapeado de texturas, etc.), hasta transformarlas en una imagen bidimensional. El conjunto de comandos necesarios para tener acceso a las funcionalidades de este hardware suele estar recogido en una **librería gráfica de bajo nivel**, la cual, además de permitir realizar una gestión óptima del hardware gráfico, actúa como substrato inicial para la implementación de otras capas software de mayor

complejidad. Como se verá en apartados posteriores, existen varias librerías de este estilo, siendo el estándar actual más aceptado la librería **OpenGl**, la cual será utilizada en los desarrollos implicados en esta tesis.

La necesidad de redibujar varias veces por segundo la escena completa, impone serias restricciones en cuanto al número de polígonos que se ven implicados en la generación de cada imagen. El coste de obtención de cada imagen, depende de varios factores **[FUN93]** relacionados directamente con el número de vértices que forman las primitivas geométricas de la escena, y el área (en pixeles) que ocupan sobre pantalla. Existen varios métodos para mantener este coste bajo un cierto límite, pero todos tienen como planteamiento básico la selección de los objetos gráficos relevantes por medio de una estructura jerárquica de dibujado conocida con el nombre de ***grafo de escena*** (*scene graph*) **[FOL91]**. Estas estructuras son árboles formados por un conjunto de nodos organizados de tal modo que cada uno de ellos ejerce una determinada influencia sobre sus ramas hijas. Los nodos hoja de este árbol usualmente están formados usualmente por primitivas de dibujado. El empleo de *grafos de escena* también facilita enormemente la organización lógica de la escena, y el control por parte del usuario.

En los siguientes apartados se muestra un estado del arte sobre el hardware gráfico, descrito a través de las funcionalidades de las librerías gráficas de bajo nivel empleadas para su manejo, se muestran las funcionalidades de las librerías gráficas de alto nivel y sus grafos de escena, y por último se realiza una revisión de los grafos de escena actuales.

2.3 Orientación hacia el Hardware Gráfico 3D. Librerías gráficas de bajo nivel.

El objetivo básico en la generación de imágenes 3D interactivas es alcanzar unas velocidades de refresco elevadas. El hecho de renderizar una escena 3D de una complejidad media varias veces por segundo, implica una inmensa cantidad de cálculos que suele sobrepasar la capacidad de la CPU del ordenador. Por ello, es norma común recurrir a la utilización de hardware gráfico especialmente diseñado para esta tarea, y, como consecuencia de un API (Application Programmer´s Interface) que proporcione un acceso eficiente a sus capacidades. Además, es deseable que dicho API proporcione una interfaz común entre los diferentes tipos de hardware gráfico existentes. En la actualidad hay varios sistemas que proporcionan un API para renderizado de gráficos 3D, pero no todos ellos resultan adecuados para su utilización en tiempo real.

Uno de los más conocidos es PHIGS (Programmers´s Hierarchical Interactive Graphics System) **[PHI88].** PHIGS está basado en GKS (Graphic Kernel System) **[GKS88]** y es un estándar ANSI que permite manipular y dibujar escenas 3D encapsulando descripciones de objetos y atributos en *display list*, esta característica es especialmente útil si existen objetos complejos que pueden ser definidos una sola vez y dibujados varias veces, o también si la escena va a ser transmitida por red. Uno de sus principales inconvenientes es que no resulta adecuada si los objetos necesitan estar actualizándose continuamente, y también que no soporta algunas características actualmente básicas tales como el mapeado de texturas.

PEXlib **[STE92]** es otro API, basado originalmente en PHIGS, pero que a diferencia de éste, permite el renderizado en *modo inmediato,* es decir, los objetos son dibujados a medida que son descritos, sin necesidad de emplear una *display list* intermedia. PEXlib presenta como inconveniente que tampoco dispone de algunas características avanzadas de rendering y que tan sólo funciona en sistemas basados en X.

Otro API muy conocido es Renderman **[UPS90]**, a diferencia de los anteriores, proporciona un lenguaje de programación para realizar las descripciones de los objetos y sus propiedades, esta característica permite generar imágenes de un nivel de acabado muy elevado, pero es muy difícil diseñar un hardware gráfico que lo soporte.

Los tres sistemas anteriores constituyen los sistemas estándar de generación de imágenes históricamente más conocidos, pero presentan algunas carencias importantes a la hora de ser empleados en la generación de imágenes en tiempo real.

OpenGl **[SEG94][NEI94][SIL95]** es un API desarrollado a partir de IrisGL **[MCL91],** una librería creada por Silicon Graphics en 1982. OpenGl tiene unas funcionalidades similares a PEXlib, pero es

independiente de X, siendo en la actualidad soportado por casi la totalidad de sistemas operativos y de hardwares gráficos.

Una alternativa a OpenGl en sistemas Windows, es el API Direct3D, el cual es un subconjunto de DirectX **[DIR02]**, una librería que además de la parte gráfica da soporte de audio, vídeo, gestión de periféricos y gestión de red. Desde sus orígenes Direct3D ha sido considerada de inferior calidad que OpenGl (incluso en sistemas Windows), sin embargo, esta visión está cambiando debido a las últimas mejoras en su API, que lo han hecho más estable y potente, y también debido a que comienza a incorporar algunas utilidades propias de un grafo de escena tales como la gestión de nivel de detalle, o la capacidad de carga de formatos 3D. Todas estas ventajas, unidas a la cercanía existente entre Microsoft y los fabricantes de tarjetas gráficas para PC esta haciendo que direct3D resulte atractivo para aquellos desarrollos que basen sus desarrollos en el sistema operativo Windows (especialmente videojuegos). Sin embargo, aunque a grandes rasgos se pueda decir que las funcionalidades como librería de bajo nivel de DirectX y OpenGl son semejantes, es necesario recordar que Direct3D no es un standard abierto, solamente funciona en sistemas Windows, y es más que dudoso que Microsoft tenga intención de hacer que DirectX esté disponible en otras plataformas.

El hecho de que OpenGl esté implementada en casi todos los sistemas operativos y sea contemplada por la gran mayoría del hardware gráfico existente, así como el hecho de que la mayoría de librerías gráficas de alto nivel, utilicen OpenGl como librería de bajo nivel para programar sus *grafos de escena*, son la causa por la cual ha sido elegida como librería base en los desarrollos implicados en esta tesis.

A continuación se muestran una serie de subapartados en los que se realiza una descripción más detallada de la librería OpenGl, así como de ciertas características que son soportadas por las librerías gráficas de bajo nivel, y que resultan de especial interés para comprender los métodos empleados en la integración de actores sintéticos en aplicaciones de gráficos en tiempo real.

2.3.1 OpenGl.

OpenGl fue introducida en 1992, y es en la actualidad un estándar industrial cuya especificación es llevada a cabo por el ARB (OpenGL Architecture Review Board), un organismo del que son socios 3DLabs, ATI, Compaq, Evans&Sutherland, Hewlett-Packard, IBM, Intel, NVidia, Microsoft, y SGI. OpenGl es multiplataforma, independiente de vendedor y abierta, lo que le permite adaptarse a las nuevas innovaciones del hardware. Soporta primitivas gráficas básicas tales como puntos, líneas, polígonos e imágenes, y operaciones tales como la aplicación de transformaciones afines o proyecciones, cálculos de iluminación, ocultación entre objetos o mapeado de texturas. En la actualidad existen gran cantidad de tarjetas gráficas desarrolladas para cumplir la especificación de OpenGl. Los drivers de OpenGl encapsulan la información sobre el hardware gráfico al que van destinados, liberando al desarrollador de aplicaciones, de tener que hacer desarrollos adaptados a las características de los diferentes tipos de tarjetas gráficas.

El funcionamiento básico de la OpenGl pude ser observado en la *Figura 2-1*. Los comandos de OpenGl son recibidos desde la parte izquierda del diagrama, la información que se envía es básicamente relativa a vértices de la geometría (Geometry Path) o a imágenes (Imaging Path). Existe la posibilidad de acumular estos comandos en *display list* para procesarlos en un instante posterior, de otro modo, los comandos son enviados directamente al pipeline de procesado.

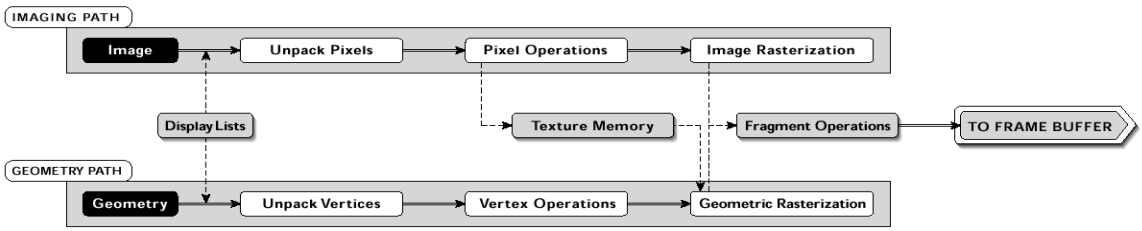


Figura 2-1. Pipeline de visualización de la OpenGl.

Tanto la información de vértices como de imagen puede llegar en un formato compactado, siendo en este caso descomprimido dentro de la propia pipeline. Puede que la información referente a la geometría esté indicada en forma curvas o superficies paramétricas, en tal caso se evalúan y transforman en una descripción basada en vértices.

En la siguiente etapa, se realizan operaciones sobre primitivas formadas de vértices (puntos, segmentos de líneas y polígonos), los vértices son transformados e iluminados, y las primitivas son ensambladas y recortadas por la pirámide de visión. También se realizan operaciones sobre las imágenes, y algunas de ellas se almacenan sobre la memoria de texturas, listas para ser aplicadas sobre la geometría 3D.

En la etapa de Rasterización, se utilizan los resultados de la fase anterior para generar una serie de direcciones y valores del *Framebuffer* mediante la utilización de una descripción 2D basada en puntos, líneas o polígonos. Por último, el contenido del *framebuffer* es actualizado teniendo en cuenta la información anterior, y un conjunto de operaciones tales como comprobaciones de *zbuffer,* operaciones de mezcla con los actuales colores de *framebuffer*, operaciones de enmascaramiento etc.

OpenGl ofrece acceso a un conjunto de operaciones gráficas del nivel más bajo posible, y como consecuencia no proporciona mecanismos directos para realizar la descripción de objetos geométricos complejos (como por ejemplo un cubo o una esfera). Estos han de ser ensamblados mediante la utilización de múltiples comandos de OpenGl.

2.3.2 Representación poligonal de los objetos.

Se puede considerar que una escena 3D está constituida por un conjunto de objetos con determinadas formas y que ocupan determinadas posiciones. La informática gráfica dispone de diferentes métodos para proporcionar definiciones matemáticas de dichos objetos, siendo una de las más comunes considerar que

la forma geométrica de un objeto está ser definido por un conjunto de triángulos conectados por sus lados (ver *Figura 2-2*).

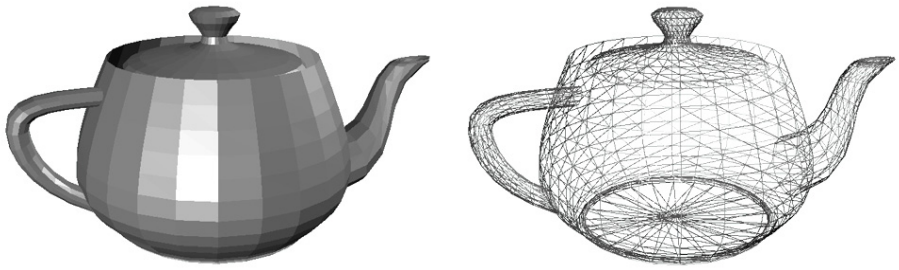


Figura 2-2. Modelo de representación de objetos 3D mediante secuencias de triángulos.

El hecho de emplear una representación de este tipo hace que el proceso de dibujado de una escena pueda ser considerado como el procesado de un conjunto más o menos grande de triángulos. Las actuales tarjetas gráficas 3D incorporan hardware capaz de procesar dichos triángulos y transformarlos en su representación correspondiente en pixeles de pantalla. Las librerías gráficas de bajo nivel proporcionan métodos para realizar descripciones de este tipo de objetos poligonales y emplear de forma adecuada el hardware gráfico.

La forma básica de descripción de un objeto en OpenGl consiste en la definición de las coordenadas que especifican vértices entre un par de comandos *glBegin*/*glEnd*, así, por ejemplo, para definir un triángulo formado por los vértices en (0,0,0) (0,1,0) y (1,0,1) sería necesaria la siguiente secuencia de código.

```
glBegin(GL_POLYGON);
        glVertex3i(0 ,0 ,0 );
        glVertex3i(1 ,1 ,0 );
        glVertex3i(1 ,0 ,1 );
glEnd();
```

Cada vértice es especificado con dos o tres coordenadas. Adicionalmente se pueden definir una normal, coordenadas de textura y color que serán utilizados en el procesamiento del vértice. La normal se utiliza en los cálculos de iluminación y es especificada por medio de un vector de 3 componentes. El color proporciona información sobre las componentes Roja, Verde, Azul y de Alfa del vértice. Las coordenadas de textura indican la forma en la que imagen que define la textura es mapeada sobre la primitiva.

En una fase previa al dibujado del triángulo se pueden haber definido otras características que afectaran a la forma en la que será dibujado, tales como las características del material empleado, la posición y características de las luces que le afectan, la presencia de niebla, etc.

2.3.3 Modelos de Sombreado. Sombreado Local.

OpenGl solamente proporciona métodos de sombreado locales. Métodos muy populares en la imagen de síntesis tradicional como son el trazado de rayos, o la radiosidad **[FOL91]**, necesitan conocer información

de otras partes de la escena diferentes al polígono que se está dibujando, y no resultan adecuados para una aplicación en tiempo real. La razón de esto es que estos métodos requieren un conocimiento global de la escena a dibujar, mientras que el hardware gráfico es una pipeline de operaciones muy localizadas, y que no tiene capacidad para almacenar y recorrer la gran cantidad de información que compone una escena compleja. Existe una posibilidad de emplear estos métodos con OpenGl que consiste en realizar un precálculo de la iluminación empleando algún método de iluminación global, y asociar los resultados a los objetos gráficos en una proceso anterior e independiente del dibujado. Este tipo de técnica solamente resulta adecuada en la representación de escenarios estáticos.

En OpenGl, el color de cada vértice es calculado de forma independiente a partir de las propiedades del material y las condiciones de iluminación. Los colores obtenidos en cada vértice son interpolados linealmente sobre las primitivas de dibujado según el método de Gouraud **[GOU71]**.

2.3.4 Primitivas de dibujado.

En una escena normal la cantidad de triángulos suele ser aproximadamente el doble que el número de vértices, esto evidencia que la gran mayoría de los vértices están siendo compartidos por varios triángulos. El procesado independiente de los vértices de cada triángulo incrementa de forma innecesaria los costes asociados a la transformación e iluminación de los vértices y también a su transferencia.

Es usual que procesado redundante de los vértices incremente estos costes en un factor de 6. Para reducir esta redundancia computacional, es común que el hardware gráfico 3D disponga de primitivas de dibujado en las que un mismo vértice es utilizado simultáneamente por varios triángulos. El tipo de primitiva de este tipo más empleada son las tiras de triángulos, también denominadas *t-mesh* o también *triangle strip*. En este tipo de primitivas, cada vértice es usado de forma conjunta con los dos vértices procesados en último lugar para generar el siguiente triángulo. En el caso de emplear *t-meshes* largas, en lugar de triángulos sueltos, la mayoría de los vértices son procesados tan sólo 2 veces, reduciendo en un factor de 3 los cálculos asociados a la transformación e iluminación de vértices. Aún así, cada vértice sigue siendo procesado 2 veces, y, por tanto, sigue existiendo un coste redundante. En la *Figura 2-3* se pueden observar las primitivas geométricas más significativas empleadas por la librería OpenGl, y como el orden en el que se definen los vértices, especifica la forma en la que se van generando los triángulos individuales.

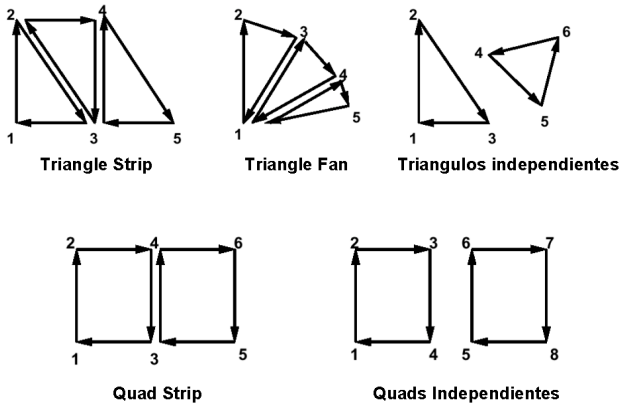


Figura 2-3. Primitivas geometrías empleadas por OpenGl.

En arquitecturas en las cuales las transformaciones geométricas e iluminación son realizadas por software, se pueden emplear estructuras indexadas en las que los vértices pueden ser procesados y almacenados una sola vez, y los triángulos se generan en una fase posterior accediendo de forma indexada a estos vértices ya procesados. Sin embargo, por razones de compatibilidad con el hardware gráfico, los APIs actuales no suelen explotar esta posibilidad.

2.3.5 Pilas de Matrices.

OpenGl proporciona métodos para realizar operaciones con matrices de 4x4, y básicamente gestiona tres tipos de ellas: La *matriz model-view* que transforma las coordenadas de los vértices; *la matriz de texturas* que es aplicada a las coordenadas de textura; y *la matriz de proyección*, la cual describe la pirámide de visión, y es empleada de forma conjunta con la matriz de model-view para transformar la escena 3D en una imagen plana. OpenGl define internamente una pila para cada uno de estos tipos de matrices. Para la gestión de estas pilas de matrices, OpenGl incorpora una serie de funciones permiten entre otras cosas definir rotaciones, traslaciones, escalados, y operaciones de *push* y *pop* sobre las pilas. La matriz en la parte alta cada pila, es la que en cada momento se está aplicando a los vértices o las coordenadas de textura.

El hecho de utilizar pilas de matrices resulta de gran utilidad en el caso de que exista una organización jerárquica de los objetos que se van a dibujar, y muy especialmente en el caso de que existan objetos con múltiples sistemas de referencia, tal y como es el caso de los actores sintéticos.

2.3.6 Empleo de Texturas.

La utilización de texturas mapeadas es una de las técnicas que más ha contribuido a mejorar la calidad gráfica de los sistemas de generación de imágenes en tiempo real. Dicha técnica consiste emplear la capacidad de asociar imágenes a las primitivas geométricas que representan los objetos, dichas imágenes proporcionan información de detalle sobre la apariencia de la superficie de los objetos, sobre sus zonas de

transparencia, permite aplicar mapas de reflejo, etc. La incorporación de esta capacidad en los sistemas de generación de imágenes en tiempo real supuso un enorme incremento en el realismo de las imágenes obtenidas (ver *Figura 2-4*).

Figura 2-4. Empleo de texturas por OpenGl.

Desde hace un par de años es común que el hardware gráfico pueda aplicar varias texturas simultáneamente sobre el mismo objeto, esta capacidad (multitexturing) ha contribuido a incrementar aún más la calidad de la imagen final.

2.3.7 Depth buffer.

El *Depth-buffer* (también conocido como Z-buffer) proporciona un método sencillo para almacenar la información sobre la profundidad sobre una imagen bidimensional. El proceso es muy sencillo, cada vez que se dibuja un pixel en pantalla, se almacena la distancia a la que dicho pixel se encuentra de la cámara. Cuando más adelante el proceso de dibujado de un objeto 3D ordene dibujar de nuevo sobre esa posición, bastará con comparar si la distancia de ese objeto es superior o inferior a la ya almacenada, si es mayor, el objeto está más lejos, y, por tanto, está siendo ocultado por el primer pixel. En este caso la orden de dibujado del nuevo pixel es anulada. En caso contrario el nuevo pixel sería dibujado, y la nueva distancia almacenada.

Normalmente cada pixel dispone de 24 o 32 bits para almacenar la información referente a la profundidad. Profundidades de *z-buffer* inferiores suelen hacer que se produzcan errores en la evaluación de la profundidad a la que se encuentran los objetos, y, por tanto, pequeños errores de dibujado.

Este método es implementado muy fácilmente en el hardware, y resulta un método para muy eficiente para calcular las ocultaciones producidas entre los objetos 3D. Existen otros métodos de ocultación que son compatibles con OpenGl, tales como la utilización de *BSP trees*, la utilización de *Octrees* o la pre-ordenación de los objetos, pero estos métodos no son soportados directamente por el hardware gráfico.

2.3.8 Shaders.

En los últimos años, los fabricantes de hardware gráfico han centrado parte de sus esfuerzos en flexibilizar la forma en la que actúa la OpenGl, permitiendo que algunas de las partes de la pipeline gráfica (hasta ahora totalmente fija) puedan ser reemplazados por unidades de código definibles por el

usuario. Estas unidades son descritas en un lenguaje de alto nivel basado en un ANSI C extendido de tal modo que es capaz de realizar operaciones con vectores y matrices, y que proporciona funciones especialmente diseñadas para su utilización en gráficos 3D. Un ***Shader*** es una unidad compilable independientemente que ha sido definida mediante este lenguaje.

2.4 Librerías gráficas de Alto Nivel. Grafos de escena.

Las librerías de bajo nivel como la OpenGl están orientadas a facilitar la utilización hardware gráfico, pero resultan insuficientes si se desea gestionar una escena compleja, y casi todas las escenas de realidad virtual lo son.

Como se ha comentado en el anterior apartado, uno de los objetivos principales de OpenGl es proporcionar un API que sea independiente del hardware gráfico, y que al mismo tiempo, permita un control total sobre sus funcionalidades. Como consecuencia de esto, OpenGl proporciona acceso a operaciones gráficas del nivel más bajo posible, y deja en manos del desarrollador de la aplicación la creación y gestión de objetos geométricos más complejos tales como cubos, esferas, coches, casas o animales. Para ayudar al desarrollador en esta tediosa tarea, existen librerías de más alto nivel que encapsulan la complejidad de estos objetos geométricos en estructuras especiales, y que, además, proporcionan métodos para gestionar la inmensa cantidad de información geométrica que suele ser manejada. Es usual que estas librerías de alto nivel, estén implementadas utilizando como base una librería de bajo nivel del estilo de la OpenGl.

Las librerías de bajo nivel proporcionan métodos para realizar únicamente descripciones geométricas muy sencillas (por ejemplo un triángulo con un material y una textura determinada). La primera necesidad de un desarrollador de aplicaciones de realidad virtual es agrupar este triángulo con otros similares para poder, por ejemplo, definir la geometría un tornillo. A su vez, agrupar este tornillo con otra serie de elementos, para formar otro objeto de mayor complejidad, como puede ser una lampara; agrupar esa lampara con otros objetos para formar una habitación, etc. Esta forma en la que los objetos del mundo real aparecen agrupados de forma jerárquica, ya podría ser una razón suficiente para que las librerías de alto nivel tomasen la decisión de emplear una estructura de datos con una organización jerárquica, pero, además, la organización jerárquica de los elementos que constituyen una escena, facilita la utilización de múltiples sistemas de referencia, y ayuda a controlar la carga del sistema. La estructura jerárquica empleada de forma generalizada por todas las librerías gráficas de alto nivel, recibe el nombre de ***Grafo de Escena*** o ***Scene Graph*** en literatura inglesa.

En los siguientes subapartados se muestran las generalidades de los grafos de escena, se presentan los tipos de nodos básicos empleados, y los métodos de selección de nivel de detalle y culling, encargados de mantener la carga del sistema dentro de unos limites razonables. Por último, se muestra la forma en la que actúan las librerías gráficas de alto nivel para aprovechar las capacidades de un sistema multiprocesador.

2.4.1 Grafos de Escena. Generalidades.

Un *grafo de escena* es un grafo dirigido acíclico de nodos que contiene los datos que definen un escenario virtual y controlan su proceso de dibujado. Contiene descripciones de bajo nivel de la geometría y la

apariencia visual de los objetos, así como descripciones de alto nivel referentes a la organización espacial de la escena, datos específicos de la aplicación, transformaciones, etc.

Los *grafos de escena* almacenan la información del escenario virtual en diferentes tipos de **nodos**. Existen nodos que almacenan la información geométrica y actúan como nodos hijos dentro del *grafo de escena*; el resto de los nodos suelen aplicar algún tipo de modificación sobre el segmento de jerarquía que depende de ellos, bien sea estableciendo agrupaciones, aplicando alguna transformación afín o realizando algún tipo de selección sobre alguna de sus ramas hijas. El proceso de dibujado consiste en realizar un recorrido de dicho grafo, aplicando las operaciones indicadas por cada tipo de nodo.

El *Grafo de Escena* tiene como funciones principales:

- Contribuir a establecer una organización lógica de la escena.
- Establecer dependencias jerárquicas entre distintos sistemas de referencia.
- Posibilitar el proceso de selección entre múltiples niveles de detalle.
- Posibilitar el proceso automático de *Culling* (eliminación automática de los objetos que se encuentran fuera del campo de visión).
- Facilitar el control de la escena por parte del usuario.
- Hacer más cómodo el acceso a las librerías gráficas de bajo nivel (OpenGl en este caso).

En la siguiente imagen (*Figura 2-1*) se puede apreciar un objeto que será empleado en una aplicación de simulación con una representación sobreimpresa del *grafo de escena* empleado para su definición. La imagen ser corresponde a una escena generada utilizando la librería *OpenGl Performer* de Sgi.

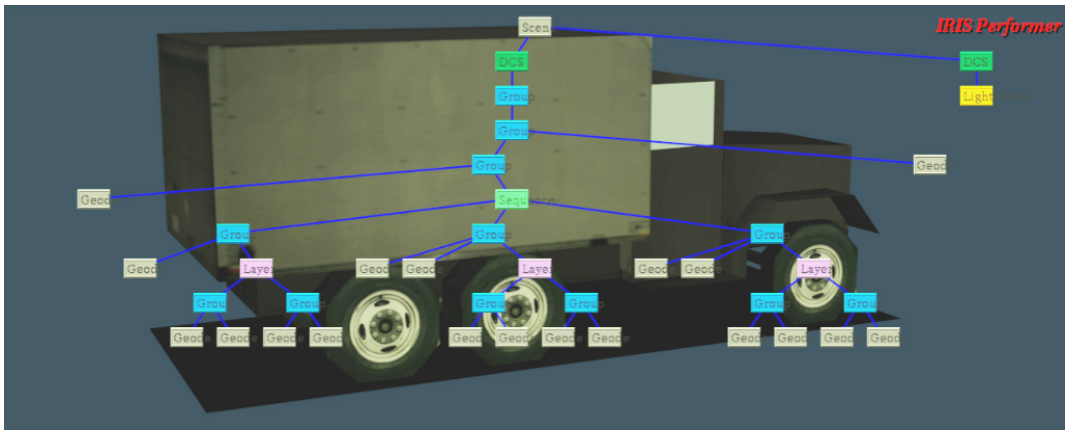


Figura 2-5. Representación de un objeto de una escena de Realidad Virtual y su correspondiente *Grafo de Escena*.

2.4.2 Tipos básicos de Nodos.

En la actualidad existen varias librerías gráficas de alto nivel, y cada uno de sus *grafos de escena* presenta sus propias particularidades. Sin embargo, existe un conjunto básico de nodos que, a veces con distintos nombres, se encuentran presentes en todos ellos:

- **Nodo de Geometría.** Almacenan la información poligonal de los objetos, también almacenan informaciones referentes a su apariencia, tales como material, textura, etc. Usualmente actúan como nodos hijo.
- **Nodo Grupo.** Se emplean para agrupar varios nodos hijos, bien sea a nivel meramente organizativo, o para facilitar el proceso de culling jerárquico.
- **Nodo Nivel de Detalle.** Usualmente llamados nodos *LOD* (Level of Detail). Seleccionan uno de sus hijos, basándose en la distancia entre el objeto con múltiples niveles de detalle y el punto de vista.
- **Nodo de Transformación Afín.** Permite aplicar una matriz de transformación que afectara a ubicación espacial de sus nodos hijos. Son necesarios para la definición de objetos móviles y también para la creación de estructuras articuladas.
- **Nodo de Switch.** Permiten realizar una selección entre sus nodos hijos.

También es usual que el usuario tenga cierta capacidad para personalizar el comportamiento de los nodos, para ello los nodos suelen tener la capacidad de almacenar datos genéricos que necesite el usuario, y también rutinas de callback escritas por el usuario que son invocadas junto con el código interno de gestión del nodo.

2.4.3 Proceso de selección de niveles de detalle.

Cualquier sistema gráfico presenta unas limitaciones que afectan al número de primitivas geométricas que pueden ser procesadas en una unidad de tiempo. Debido a estas limitaciones, el objetivo principal de las bases de datos orientadas a tiempo real, es emplear estrategias que consigan reducir la complejidad de la base de datos a representar, sin que ello repercuta en una merma de la calidad visual final. La selección de niveles de detalle es una de estas estrategias.

La idea principal del proceso de selección de niveles de detalle se basa en la observación de que un objeto, cuando se encuentra alejado, no puede ser apreciados en detalle, bien sea por el reducido número de pixeles que ocupa en pantalla, debido al efecto de la perspectiva, o bien por efecto de las condiciones atmosféricas, tales como la niebla, o la iluminación. Esta circunstancia hace que un objeto originalmente muy complejo, pueda ser sustituido por otro geométricamente más sencillo. Como consecuencia, una base de datos para gráficos en tiempo real suele hacer que cada objeto presente distintas versiones de diferente complejidad geométrica, cada una de las cuales representa al objeto en un margen de distancias concreto **[JON96]**. Además, se ha de establecer un criterio que determine cual de ellas en concreto ha de ser

visualizada en cada instante. El criterio suele consistir en emplear la distancia existente entre el objeto y el punto de vista, en función de ella, se selecciona la representación que resulte más adecuada: una representación geométricamente muy sencilla si el objeto se encuentra tan alejado que es imposible percibir sus detalles, o una estructura geométricamente compleja si el objeto se encuentra tan cercano que sus detalles resultan perceptibles.

Si bien la distancia entre el objeto y el punto de vista, es el criterio principal para realizar la selección entre los distintos niveles de detalle, en realidad, para que el proceso se realice de forma optima, se han de tener en cuente otros factores tales como el campo de visión empleado, la resolución en pixeles de la imagen, la precisión óptica del sistema de monitorización, ciertos efectos atmosféricos como la niebla o el humo, etc. Otros aspectos que pueden ser tenidos en cuenta, son la importancia del objeto dentro de la escena, o la carga del sistema.

Es usual que cada objeto presente tres o más niveles de detalle. En general, el hecho de disponer de varios niveles de detalle intermedios hace las transiciones entre niveles de detalle resulten menos bruscas. La condición de cambio de nivel de detalle, (en función de la distancia u otros criterios) se produce en un instante concreto, y esto hace, que las representaciones de un objeto, entre un frame y el siguiente, resulten ligeramente diferentes. El sistema visual humano es muy sensible a estas discontinuidades, y es necesario prestar un interés especial a la forma en la que se producen los cambios entre niveles de detalle. Existen varias técnicas, que permiten reducir los efectos indeseables de las transiciones entre niveles de detalle, unas de ellas consisten en seleccionar el punto adecuado de transición, otras, en la creación de muchos niveles de detalle distintos, de tal forma que las diferencias entre dos niveles de detalle consecutivos sean mínimas, y otras hacen que los saltos no se produzcan de forma brusca, sino mediante transiciones suaves entre las representaciones, no existiendo una distancia a la cual un modelo es directamente sustituido por otro, sino un rango de distancias, o bien, un rango temporal en el cual se realiza una transición progresiva entre las dos representaciones: La técnica de *Fading* produce un fundido suave entre las imágenes correspondientes a dos niveles de detalle consecutivos, pero, presenta como inconveniente que incrementa temporalmente la carga en el periodo en el que los dos modelos están siendo dibujados simultáneamente; la técnica de *Morphing,* hace que los vértices de la geometría de uno de los modelos se modifiquen suavemente, hasta adoptar la posición de los vértices del otro modelo. Esto requiere que algunos vértices sean creados o destruidos en tiempo real, y, por tanto, obliga a que existan fuertes dependencias entre los modelos que representan los distintos niveles de detalle. Durante los periodos de transición la CPU consume un tiempo adicional en la gestión del *morphing*. Es el método que presenta mejores resultados tanto a nivel visual como computacional, sin embargo, el hecho de que existan dependencias tan fuertes entre los distintos niveles de detalle, hace que dichos modelos sean muy difíciles de generar, y que en la practica, sea muy poco usado.

Para tener una referencia de la magnitud de la reducción de cálculos que supone la utilización de niveles de detalle, se puede indicar que la representación poligonal de una persona de una calidad aceptable

requiere unos 2250 polígonos **[KOE97]**, mientras que el nivel de detalle más bajo puede ser resuelto con tan sólo 38.

En la *Figura 2-6* se pueden apreciar los diferentes niveles de detalles empleados en la representación del sistema de rodadura de un tren.

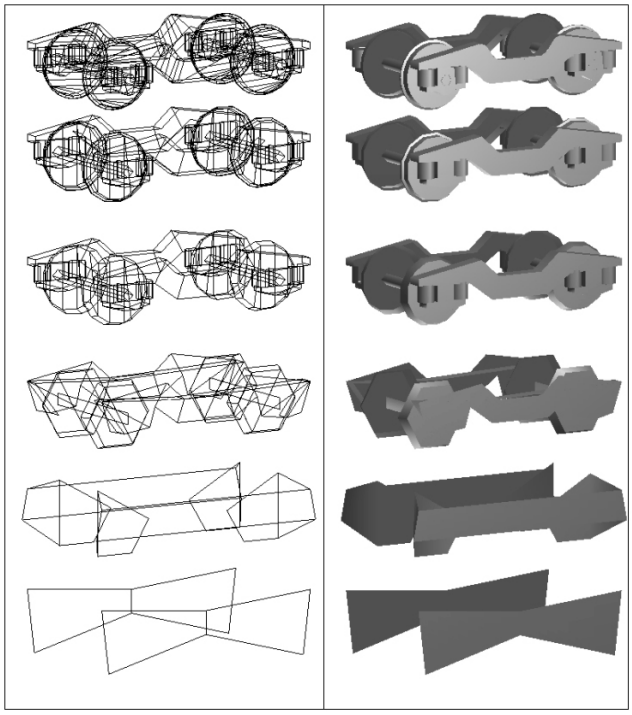


Figura 2-6. Diferentes niveles de complejidad en la representación geométrica del sistema de rodadura de un tren.

En la figura anterior, la representación de más calidad tiene aproximadamente unos 1200 vértices, la que aparece en cuarto lugar presenta aproximadamente 120 vértices, y por último la más sencilla, está formada por tan sólo 12 vértices. Esto evidencia que con la utilización adecuada de niveles de detalle se obtienen unas tasas de reducción de su coste computacional que se mueven en dos ordenes de magnitud.

En la *Figura 2-7* se ha realizado una representación conjunta de las geometrías que representan el tercer y cuarto nivel de detalle, situándolas a distancias de la cámara cada vez mayores, es fácil comprobar que a medida que la distancia va aumentando las diferencias entre ellos se vuelven imperceptibles.

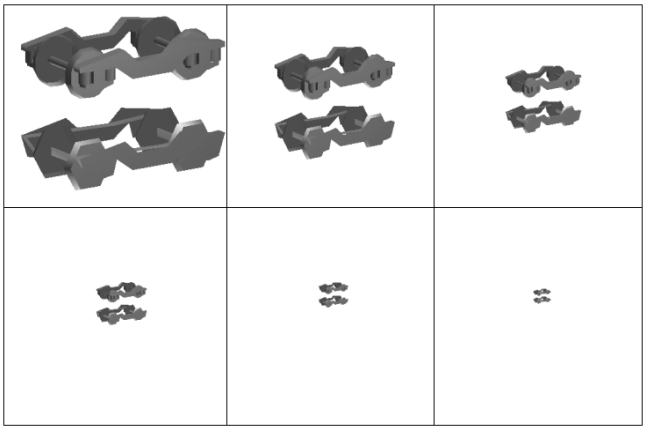


Figura 2-7. Evidencia de que la capacidad de percepción de detalles disminuye con la distancia del objeto

Existe gran número de investigaciones relativas al desarrollo de técnicas de simplificación geométrica que pueden ser empleadas para la generación de niveles de detalle. **[TUR92] [LOR93] [ROS93][HOP93]**.

2.4.4 Proceso de culling.

En una aplicación de simulación, o en cualquier entorno virtual en general, el usuario se encuentra inmerso en una inmensa base de datos geométricos. Sin embargo, en cada instante está mirando en una dirección concreta, y, por consiguiente, gran parte de la geometría permanece fuera de su campo de visión. Enviar toda la base de datos al hardware gráfico resultaría muy costoso, y, además, es innecesario. La CPU puede determinar cuales son los objetos que están dentro de la pirámide de visión (denominada *viewing frustum* o simplemente *frustum* en la literatura inglesa) y con ello reducir considerablemente la cantidad de datos a transferir al hardware gráfico y consecuentemente la cantidad de trabajo a realizar por éste. Este proceso es conocido como **culling**, y es junto el proceso de gestión de niveles de detalle, uno de los principios que permite mantener la cantidad de geometría a dibujar dentro de unos márgenes adecuados

Existen métodos de culling muy avanzados que tienen en cuenta las ocultaciones producidas entre los objetos de la escena, este tipo de culling es esencial en el caso de entornos de tipo arquitectónico, formados por muchas habitaciones y paredes. Básicamente consisten en determinar que objetos son visibles a través de las puertas y ventanas **[AIR90][TELL92],** y almacenar la información en estructuras de datos adecuadas que son rellenadas en una fase previa a la simulación. Otro tipo de técnica de *culling* consiste en dividir el espacio en celdas regulares, y calcular que otras celdas son visibles desde cada una de ellas. Esta operación ha de ser también realizada en una fase de preproceso. En tiempo de ejecución simplemente se determina la celda en la que está el usuario, y en función de ello, se activa o desactiva la visualización de celdas correspondientes. Esta técnica es especialmente útil en aplicaciones en las que el espacio puede ser fácilmente compartimentado, como por ejemplo aquellas que consisten en el recorrido de carreteras o túneles **[JOH94][BAY95]**.

Estas dos últimas técnicas pueden ser aplicadas en bases de datos que tienen características muy concretas, y presentan como ventaja principal que los cálculos importantes son realizados en una fase de preproceso. Sin embargo, no resultan totalmente adecuadas en aquellos casos en los que la base de datos contiene objetos móviles, o bien, en aquellos casos en los que el usuario no tiene ningún tipo de restricción en cuanto a su movimiento.

Existen métodos de culling genéricos que pueden ser aplicados en cualquier circunstancia, y que están basados en el cálculo de intersección entre la pirámide de visión y las envolventes de los objetos de la escena (conocidos en la literatura inglesa como ***bounding volumes***). Esta técnica presenta como inconveniente principal, el hecho de que las intersecciones han de ser calculadas en tiempo de ejecución. Los algoritmos que realizan este tipo de culling han de ser capaces de establecer un balance entre la carga que producen sobre la CPU, y el ahorro de trabajo que producen sobre el hardware gráfico.

La utilización de *bounding volumes* resulta especialmente adecuada para las bases de datos organizadas en forma de *Grafo de Escena*, ya que su estructura puede ser aprovechada para definir una organización jerárquica de *bounding volumes*. Cada nodo tiene asociado un *bounding volume* que engloba al nodo en sí, y a todos sus hijos. La librería que gestiona el *scene graph,* tiene capacidad para recalcular automáticamente la forma de esas envolventes cada vez que existe una modificación en la topología de la base de datos. Es usual que cada nodo tenga una lista que contiene referencias a sus nodos padre, de modo que cuando el *bounding volume* de un nodo es modificado, esa modificación es propagada hacia sus padres. De este modo la jerarquía de *bounding volumes* permanece continuamente actualizada.

Es bastante común emplear como envolventes paralelepípedos alineados con los ejes (también conocidos como ***bounding boxes***), o esferas (***bounding spheres***). Este tipo de envolventes son aproximaciones de la forma original, y, por tanto, puede ocurrir que el *bounding volume* de un objeto esté intersectando con la pirámide de visión, y, sin embargo, no ocurra lo mismo con el objeto en sí. Por esta razón, es adecuado que la envolvente se adapte lo máximo posible a la forma real del objeto. Analizado desde esta óptica, las *bounding boxes* son una mejor aproximación, sin embargo, la actualización, transformación y testeado mediante *bounding spheres* es mucho más rápida. Algunas librerías como OpenGl Performer emplean sistemas mixtos, en los que las geometrías finales del *grafo de escena* se procesan mediante *bounding boxes* y los nodos intermedios mediante *bounding spheres*.

El método de procesado de una escena con este tipo de *culling* jerárquico consiste en hacer un recorrido recursivo del *grafo de escena*, comprobando para cada nodo la forma en la que su envolvente intersecta con la pirámide de visión, tomando las siguientes decisiones:

- El *bounding volume* está completamente fuera de la pirámide de visión: ni este nodo ni sus hijos han de ser dibujados. El recorrido del *scene graph* continúa sin entrar en esta rama.
- El *bounding volume* está completamente dentro de la pirámide de visión: este nodo y sus hijos han de ser dibujados. En esta rama ya no es necesario realizar más comprobaciones de *cull*.

- El *bounding volume* intersecta con la pirámide de visión: se continúa realizando el proceso de *cull* sobre los nodos hijos. En el caso de que sea un nodo final, se procede a su dibujado.

En el caso de emplear este tipo de *culling* es muy conveniente tener una buena organización jerárquica de la escena, de tal modo que los objetos con una ubicación espacial próxima aparezcan agrupados debajo de un mismo nodo. Así por ejemplo, supongamos en el caso del tornillo que forma parte de una lampara, la cual se encuentra en el salón de una casa que forma parte de un pueblo; si esta escena tuviese una organización jerárquica adecuada en la que hubiese sucesivos nodos tornillo->lampara->salon->casa->pueblo, bastaría con comprobar que la casa está fuera de la pirámide de visión para descartar su dibujado y las comprobaciones de culling con todos los objetos que se encuentran en su interior. En siguiente diagrama (*Figura 2-8*) se muestra de forma gráfica una situación similar en la que se representa una escena sencilla formada por seis objetos que aparecen agrupados mediante distintos nodos intermedios, así como sus correspondientes *bounding spheres.* Se puede observar claramente la relación entre la organización jerárquica de los nodos del grafo de escena y las *bounding spheres* correspondientes a cada nodo.

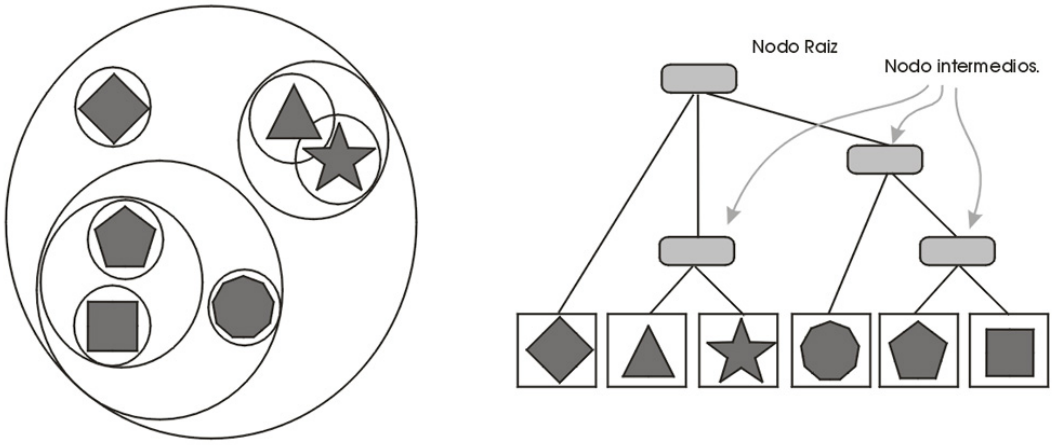


Figura 2-8. *Bounding volumes* en un proceso de *culling* jerárquico.

Existen métodos de culling que también tienen en cuenta las condiciones de visibilidad debidas a las condiciones atmosféricas (bien sea niebla, lluvia, etc.). Un objeto que normalmente es visible a una determinada distancia puede, que a la misma distancia, deje de serlo en el caso de que exista una niebla densa.

2.4.5 Multiproceso.

Las aplicaciones de gráficos en tiempo real suelen consumir gran cantidad de recursos, por ello es habitual que en algunos casos sea necesario recurrir al empleo de ordenadores con varios procesadores.

Hay dos formas posibles de emplear de forma conjunta varios procesadores: La organización en paralelo y la organización en serie o pipeline. La primera de ellas presenta como ventaja que tiene poca latencia,

pero es de difícil implementación, pues encontrar la forma en la que paralelizar procesos encierra bastante complejidad. La organización en pipeline, presenta como inconveniente que tiene más latencia que la anterior, pero, por el contrario, resulta de una implementación mucho más sencilla. También es posible definir estructuras mixtas, pero es un área que no ha sido muy investigada.

2.4.5.1 Procesamiento Paralelo:

Hay varias formas de paralelizar un algoritmo **[CULL98]**: empleando asignación estática, el trabajo total es dividido en varios bloques de trabajo, y cada procesador realiza uno de dichos bloques de trabajo en paralelo con los otros. Cuando los procesadores han terminado de procesar sus bloques, el resultado ha de ser mezclado, para que esto sea factible, es necesario que el coste computacional del procesado de cada bloque sea altamente predecible. Mediante la asignación dinámica, los trabajos a realizar por las CPUs son dejados en una zona común, cuando un procesador ha finalizado su trabajo actual toma otro bloque de trabajo de la zona común y lo procesa. El hecho de que cada CPU solicite un paquete de trabajo, supone una sobrecarga que se puede minimizar haciendo que los paquetes de trabajo tengan un tamaño adecuado. Esto puede evitar que el sistema se quede desbalanceado debido a que alguna CPU tenga que estar esperando a que finalizase el trabajo alguna otra que había tomado un bloque de trabajo demasiado grande.

El principal problema de la utilización de un procesamiento paralelo es que necesita que el proceso pueda ser dividido en paquetes de trabajo que puedan ser realizados de forma paralela.

2.4.5.2 Procesamiento Serie o en Pipeline.

La forma de realizar un proceso en serie consiste en dividir el trabajo en una serie de bloques que pueden ser realizados en cascada, de tal modo que los datos de salida de un procesador son empleados como entrada del siguiente.

La generación de una imagen en tiempo real puede ser considerada como constituida por una serie de etapas que se realizan en cascada. La forma tradicional de emplear varios procesadores en una aplicación de gráficos en tiempo real consiste en dividirla en tres etapas **[ROH94]**: APP, CULL y DRAW. El procesado de la escena se realiza en el proceso de APP, y consiste en la actualización de los valores de los nodos con los valores provenientes de algún mecanismo de gestión del comportamiento de la escena. El procesado visual consiste en la realización de las operaciones de culling y dibujado (CULL y DRAW), es habitual que los procesos de *culling* y *draw* sean realizados de forma automática. La misión de cada una de estas etapas es la siguiente:

- **APP**. Lee valores de los periféricos de entrada, realiza la simulación de los objetos que se mueven, actualiza la database visual e interacciona con otras posibles simulaciones conectadas a través de la red.
- **CULL.** Aplica el culling jerárquico determinando que porciones de la escena son potencialmente visibles, realiza una selección de los niveles de detalle, realiza una ordenación de la geometría por su

estado gráfico, ordena los objetos transparentes por distancia para su correcto dibujado, y finalmente, crea una lista con los objetos que han de ser dibujados.

- **DRAW**. Se encarga de alimentar al hardware gráfico con la lista de los objetos que han quedado de la fase anterior.

Algunas librerías gráficas como OpenGl Performer incluyen en la pipeline de procesado de la escena otros procesos:

- **ISECT.** Realiza cálculos de intersección entre segmentos de línea y la geometría del *grafo de escena,* lo cual se puede emplear para detección de colisiones.
- **COMPUTE**. Realiza cálculos complejos de forma asíncrona.
- **DBASE**. Gestiona de forma asíncrona la base de datos que define la escena.

Dependiendo del número de CPUs disponibles y del tipo de aplicación se pueden emplear distintos modelos de multiproceso, así, en el caso de disponer de sólo un único procesador las tres etapas serían ejecutadas en la misma CPU. En el caso de disponer de dos, los procesos de APP y CULL pueden ser ejecutada en una y el DRAW en la otra, o bien se ejecutaría la etapa de APP en una CPU y las de CULL y DRAW en las otra. En el caso de existir 3 procesadores, uno podría estar dedicado al APP, otro al CULL y otro al DRAW. En sistemas de simulación complejos con dos o más salidas gráficas, es común encontrar estructuras mixtas formadas por 8 o más CPUs.

2.5 Revisión de *Grafos de Escena*.

Una vez presentadas las características básicas de las librerías gráficas de bajo y alto nivel, resulta adecuado analizar con más profundidad las características de los *grafos de escena* comerciales más conocidos. Si bien en este apartado no se profundiza en el análisis de la totalidad de los *grafos de escena* actuales, sí que se trata sobre aquellos que han tenido una mayor influencia en la evolución de esta rama de la informática gráfica. Con esta intención se hace una descripción del funcionamiento de *OpenGl Performer*, que encapsula en estructuras de más alto nivel la mayoría de las funcionalidades de OpenGl y proporciona funcionalidades añadidas orientadas hacia aplicaciones que necesitan una elevada tasa de refresco, como por ejemplo aplicaciones de simulación visual o escenografía virtual; la librería *Open Inventor* que proporciona una interfaz gráfica y un *grafo de escena* muy flexibles que facilitan la creación y manipulación interactiva de una escena 3D; y por último se profundizara en el *VRML*, un formato de fichero que describe un *grafo de escena* y que tiene la intención de ser el estándar para la navegación en espacios tridimensionales a través de Internet.

En los siguientes apartados se detallaran las propiedades básicas de estos *grafos de escena,* y se hará una descripción de cuales son los nodos que los constituyen y características. En un último apartado se muestran de forma rápida otros grafos de escena existentes en la actualidad.

2.5.1 OpenGl Performer.

OpenGl Performer es un Grafo de Escena diseñado para optimizar al máximo la utilización del hardware gráfico **[ROH94]**. Es una librería muy orientada al mercado de la simulación visual, siendo su mayor prioridad la obtención de tasas de refresco muy altas, llegando a mantener tasas constantes de 50 frames/segundo. Proporciona soporte para multiproceso, permitiendo dividir el trabajo entre múltiples CPUs, gestionando su sincronización y transferencia de información. Además, proporciona algunas características avanzadas tales como la detección de intersecciones, la gestión de terrenos, morphing geométrico o incluso una pequeña interfaz de usuario.

Los nodos utilizados por la librería Performer son los siguientes:

- **Nodo Geode:** Es un nodo hoja que almacena la información geométrica de un objeto (**Geo**metry no**de**). Está formado por una lista de estructuras llamadas *pfGeoSet* que encapsulan características de bajo nivel referentes a primitivas de dibujado y su apariencia gráfica, y que almacenan la información en un formato muy próximo a la OpenGl.
- **Nodo Grupo:** Sirve para agrupar varios nodos hijos.
- **Nodo Scene:** Actúa como nodo padre de todo el *grafo de escena*. Se utiliza también para almacenar informaciones visuales comunes a toda la escena, tales como condiciones de iluminación, modos de dibujado, condición de niebla, etc.

- **Nodo SCS:** Permite definir un sistema de coordenadas que no pueden ser modificados durante la simulación (Static Coordinate System), se utilizan para ubicar objetos en distintas posiciones de la escena
- **Nodo DCS:** Permite definir un sistema de coordenadas que puede ser modificado durante la simulación (Dynamic Coordinate System). Se utiliza para implementar objetos articulados u objetos que se desplazen por la escena.
- **Nodo FCS:** (Flux Coordinate System). Es de características similares al DCS, pero presenta características especiales para su utilización en multiproceso.
- **Nodo Switch:** Es un nodo que puede tener varios hijos y permite que el usuario seleccione cuales de ellos quiere que sean dibujados.
- **Nodo Secuence:** Es un nodo que puede tener varios hijos, los cuales va mostrando de forma secuencial. Se utiliza para representar secuencias animadas. Una secuencia consiste en una lista ordenada de hijos, cada uno de los cuales con una duración asignada. Es posible hacer que la secuencia se ejecute de inicio a fin, de fin a inicio, que se repita cíclicamente, etc.
- **Nodo LOD:** Se emplea para gestionar distintos niveles de detalle de un objeto.
- **Nodo Layer:** Es un nodo hoja, que permite establecer un orden de dibujado en el caso de geometría coplanar.
- **Nodo LighSource:** Contiene la especificación de una fuente de luz.
- **Nodo BillBoard:** Rota una geometría de modo que siempre aparezca orientada hacia el punto de vista. Es especialmente útil para representar objetos que tienen una simetría axial, como pueden ser árboles, farolas, etc.
- **Nodo Partition:** Particiona la geometría para realizar intersecciones eficientes.
- **Nodo Text:** Es también un nodo hoja que se utiliza para renderizar textos tridimensionales.
- **Nodo ASD:** Permite realizar transiciones suaves entre superficies complejas tales como grandes superficies de terreno (Active Surface Definition).

2.5.2 Open Inventor.

La librería Inventor es también un sistema de definición, manipulación y renderizado de escenas 3D basada en la utilización de descripciones geométricas de alto nivel **[STR93].** La versión original se denominaba Iris Inventor funcionaba exclusivamente sobre máquinas SGI, y estaba implementada como una capa por encima de la librería IrisGl. En el momento que ARB creo la especificación de librería OpenGl, también se realizó una descripción del Open Inventor, una versión multiplataforma del antiguo Iris Inventor que utilizase como base a la librería OpenGl.

Inventor prima la usabilidad sobre el rendimiento. Está formado por un conjunto muy elaborado de nodos de uso muy sencillo, pero no proporciona un buen rendimiento en tiempo real (al menos comparado con otros sistemas como OpenGl Performer).

Las principales características de Open Inventor son:

- Facilita la organización de escenas 3D. Por tener una estructura de *grafo de escena*, permite organizar jerárquicamente la información de forma sencilla.
- Proporciona estructuras gráficas predefinidas. Las formas básicas proporcionadas por Inventor son cajas, conos, esferas y cilindros, además, permite definir textos 2D y 3D.
- Proporciona utilidades gráficas ya definidas, por ejemplo rutinas para manejar matrices.
- Tiene mecanismos muy flexibles de descripción de formas y objetos. Permite definir objetos a partir de curvas y superficies de tipo NURBS, y también a partir de mallas de triángulos.
- Tiene capacidades de rendering sofisticadas. Permite seleccionar entre distintos modos de renderizado. Se pueden seleccionar distintas cámaras y distintos modos de visualización interactiva.
- Proporciona un método integrado de selección y manipulación interactiva de objetos.
- Es extensible de tal modo que es posible crear nuevos tipos de primitivas y objetos.
- Tiene un formato de fichero 3D con descripciones de alto nivel.
- Permite incorporar animaciones, teniendo nodos especialmente diseñados para ese propósito.
- No está diseñado para soportar multiproceso.

Inventor es un *grafo de escena* en el que las propiedades de cada nodo afectan a los nodos que están por debajo de él y también a los nodos colocados a su derecha. Cada nodo puede ser una forma, una propiedad de su apariencia, una transformación, una cámara o una luz. Algunos de los tipos nodos que Open Inventor emplea son los siguientes:

- **Nodo Shape:** Representa un objeto 3D o 2D como puede ser una esfera o un cubo.
- **Nodo Transform:** Representa transformación afín, como por ejemplo una traslación o una rotación.
- **Nodo Appearance:** Modifica la apariencia de los objetos que le siguen en el grafo.
- **Nodo Light:** Aplica una iluminación a los nodos que le siguen.
- **Nodo Camera:** Visualiza los nodos que le siguen.
- **Nodo Separator:** Separa el efecto de los nodos que hay por debajo de él de tal modo que no afecte a los nodos que hay a su derecha.

- **Nodo Units:** Especifica el tipo de unidades medida del mundo real que se corresponde a los nodos que le siguen (como por ejemplo centímetros, pulgadas o metros), y aplica un factor de escala en el caso que sea distinto de los nodos que le precedían.
- **Nodo Sensor:** Detecta el momento en el que ocurre un determinado evento, como por ejemplo que un *timer* llegue a su final, o que un nodo haya sido seleccionado por el usuario, y lanza una llamada a una rutina de callback definida previamente.
- **Nodo Manipulator:** Asigna a los objetos una interfaz 3D que permite de modificar su posición, su tamaño y orientación de un modo muy sencillo
- **Nodo Complexity:** Permite controlar la calidad con la que se quieren dibujar los nodos que le siguen.
- **Nodo Texture:** Especifica una textura para los nodos que le siguen.
- **Nodo Environment:** Define efectos atmosféricos, como por ejemplo niebla.
- **Nodo Normals:** Es utilizado para realizar cálculos de iluminación.

En la *Figura 2-9* se puede apreciar el aspecto de un *Grafo de Escena* de *Inventor*. Todos los *grafos de escena* comienzan con un nodo que actúa como raíz, y usualmente existe una cámara en la parte izquierda del grafo que es el que permite visualizar el resto de la escena. El siguiente nodo es de tipo *Light*, que sirve para iluminar a los objetos que le siguen, si ese nodo estuviese puesto después de la primera esfera, ésta no estaría iluminada. El resto son los objetos de la escena. Cada uno de ellos tiene su propio subgrafo que comienza por un nodo *Separator* que hace que lo que ocurra en el subgrafo no afecte al resto de los nodos. A cada nodo *Shape* le precede un nodo *Transform*, que permite colocar cada objeto con la posición y orientación deseada.

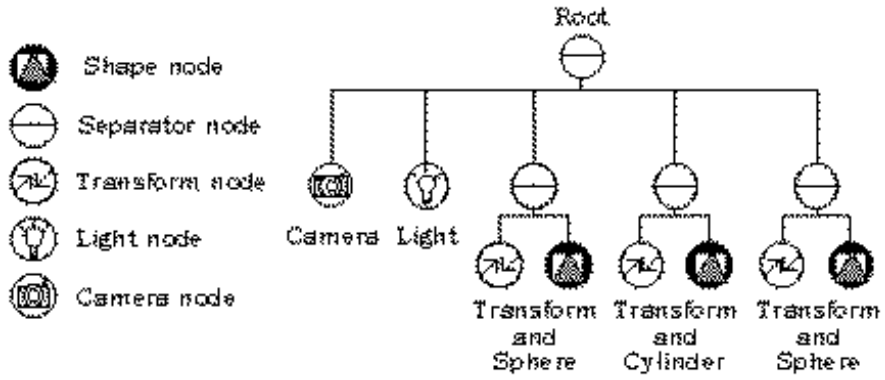


Figura 2-9. Ejemplo de Grafo de escena utilizado por Inventor.

Una de las objetivos principales de Inventor es definir escenas 3D interactivas. Inventor gestiona automáticamente la selección y manipulación de objetos 3D, así como el movimiento el usuario por la por la escena 3D o en torno a los objetos 3D. También permite asociar animaciones automáticas a los objetos.

2.5.3 VRML.

VRML (Virtual Reality Modeling Lenguage) es un formato de fichero de texto que sirve para realizar descripciones de objetos 3D y entornos interactivos en Internet. Permite la incorporación de textos, imágenes y secuencias de audio y vídeo. Es posible hacer que un entorno VRML, conecte con otro entorno VRML en otro lugar de la red de dos formas posibles: puede que un objeto 3D de una escena, por ejemplo un mueble, se esté cargando de una dirección en Internet diferente de la de la habitación principal, o puede que al seleccionar la puerta de dicha habitación accedamos a una habitación que está en otro servidor dentro de la red. Además, VRML ha sido diseñado para trabajar interactuando con Java y JavaScript. Concebido como una extensión de Inventor, sufre de sus mismos problemas de rendimiento.

El formato VRML se encuentra en su versión 2.0. Su primera versión (VRML 1.0) sólo permitía crear y visualizar mundos 3D estáticos. En la versión VRML 2.0 incorpora elementos que permiten hacer que los objetos 3D se modifiquen automáticamente y sean capaces de responder a la acción del usuario.

En la *Figura 2-10* se muestra un esquema con los definidos por VRML, gran cantidad de ellos (Group, switch, LOD...) ya son conocidos de grafos de escena como Performer o Inventor.

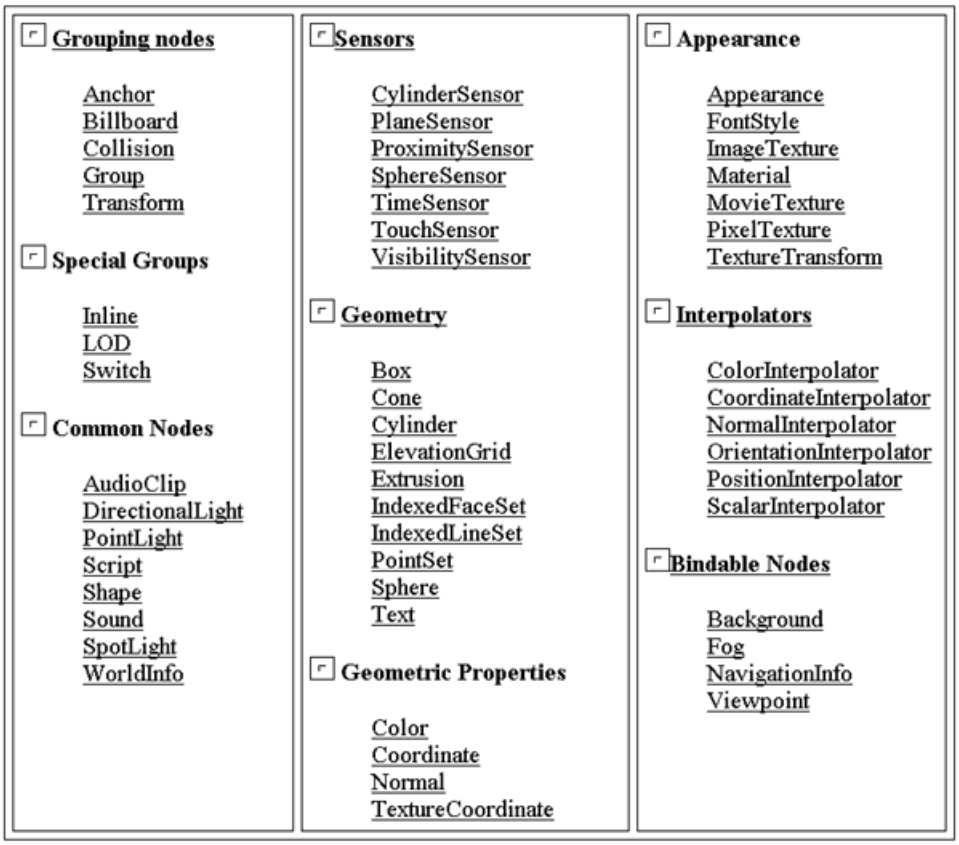


Figura 2-10. Nodos de VRML 2.0.

Llama la atención de existencia de nodos muy especializados como el *Background* que permite añadir imágenes de fondo, lo que permite representar por ejemplo montañas lejanas y un cielo, o de un tipo de nodo para crear fácilmente terrenos irregulares (nodo *ElevationGrid*). También existen nodos como el

Collision que permiten hacer que un objeto reaccione como si fuese sólido de tal modo que el usuario no puede atravesarlo y sí caminar sobre él. Existe un grupo de nodos que actúan como *sensores* (de un modo similar a como hacia inventor) y que permiten que la escena reaccione a las acciones del usuario. Los distintos nodos de tipo *interpolator* permiten crear animaciones predefinidas que pueden ser ejecutadas cuando sea necesario. Con este tipo de nodos se pueden definir modificaciones dinámicas sobre los objetos de la escena, así es posible definir un objeto que presente un morphing geométrico, una puerta de apertura automática o un objeto que varia cíclicamente su color o que recorre una trayectoria determinada. El nodo *Sound* permite incorporar sonidos con una ubicación espacial tridimensional. Uno de los nodos más importantes en VRML es el nodo *Script*, el cual básicamente contiene un programa que recibe eventos de entrada, los procesa y genera eventos nuevos, mediante la utilización de este tipo de nodos se puede hacer que la escena y sus objetos presenten comportamientos muy sofisticados. También resulta muy interesante la capacidad que tiene el usuario para definir nodos PROTO los cuales pueden contener en su interior más nodos VRML, campos y scripts. Un desarrollador puede definir un nodo PROTO con unas determinadas características, de tal modo que puede ser utilizado, modificado o ampliado por un segundo desarrollador.

En general se puede establecer una clasificación similar en nodos grupo y nodos hoja similar a la de Performer o Inventor. Pero la gran cantidad de nodos (*Figura 2-10*), y las interdependencias que se establecen entre ellos, hacen VRML resulte mucho más confuso que los *grafos de escena* anteriores.

2.5.4 Otros.

En los últimos años han surgido distintos grafos de escena, es norma general que de un modo u otro reproduzcan las funcionalidades de los tres citados anteriormente. Cabe destacar que algunos de ellos proporcionan un rendimiento similar a OpenGl Performer y son de código abierto y multiplataforma. No tiene sentido aquí profundizaren las particularidades de cada uno de ellos, simplemente se mostrará una descripción abreviada y un enlace de Internet para obtener información adicional:

OpenRM. (http://openrm.sf.net)
Es un entorno de desarrollo OpenSource, empleado para implementar aplicaciones 3D en sistemas Unix/X11 y Win32, puede trabajar con varios threads.

Open Scene Graph .(http://www.openscenegraph.org)
Es un API que proporciona una rica jerarquía de clases. Proporciona un grafo de escena que Open Source e independiente de plataforma. Incorpora una interfaz que permite acceder a una librería, de nombre Cal3D, que permite renderizar en tiempo real personajes con piel generados a partir de programas de modelado como el 3D Studio Max. Es capaz de renderizar entre 20 y 30 actores en tiempo real, lo cual lo convierte en una opción adecuada para muchos tipos de aplicaciones prácticas y videojuegos.

OpenSG. (http://www.opensg.org)

Es una librería basada en C++ y OpenGl que contiene un grafo de escena. Es distribuido bajo los principios de Open Source, es multi-thread, multiplataforma (Linux, Irix, Windows), y fácilmente extensible. Es mantenido por varias empresas y centros de investigación.

PLIB/SSG. (http://plib.sf.net).

PLib es un conjunto de librerías multiplataforma diseñadas para ayudar a los desarrolladores de aplicaciones 3D interactivas. Es también distribuida bajo los principios de OpenSource. SSG (Simple Scene Graph) es uno de sus módulos que proporciona un grafo de escena implementado sobre la base de OpenGl y C++.

RMScenegraph. (http://www.r3vis.com/RMSSceneGraph)-

Es un grafo de escena basado en OpenGl, actualmente disponible para Unix, Linux y Win32.

SGI OpenGL Optimizer. (http://www.sgi.com/software/optimizer).

Es un conjunto de herramientas orientadas al mercado del CAD/CAM. Optimizer en sí mismo no es un grafo de escena, pero emplea internamente un grafo de escena (similar a VRML) especialmente optimizado para manejar modelos voluminosos.

SGL. (http://sgl.sf.net).

Es un conjunto de librerías multiplataforma desarrolladas en C++ y que operan sobre OpenGl, gestiona de forma adecuada la selección de objetos y cálculo de intersecciones, permite cargar distintos tipos de formatos de imagen, y dispone de varios métodos de culling.

Sun's Java 3D. (http://java.sun.com/products/java-media/3D).

El API de Java3D proporciona un conjunto de interfaces orientados a objetos, que permiten a los desarrolladores definir una escena de 3D y su comportamiento, de una forma independiente de plataforma. Internamente dispone de un grafo de escena que introduce algunos conceptos comúnmente no considerados como parte del entorno de simulación tales como sonido espacial 3D.

2.6 Conclusiones.

En este capítulo se han mostrado las diferencias entre la informática gráfica tradicional y la informática gráfica en tiempo real, mostrando la necesidad de utilización de hardware especializado en la generación de imágenes, y en consecuencia, de librerías de bajo nivel que proporcionen control sobre dicho hardware. En este sentido, se ha realizado un recorrido histórico por las distintas librerías que proporcionan un API para el renderizado de gráficos 3D, finalizando con la librería OpenGl, un standard empleado por la gran mayoría de los sistemas gráficos actuales. Se ha presentado de forma abreviada los principales aspectos que definen la forma de funcionamiento de la librería OpenGl, realizando también una introducción sobre la forma en la que opera con las matrices de transformación.

Se ha descrito como las librerías de bajo nivel proporcionan buenos métodos para controlar las capacidades del hardware gráfico, pero resultan demasiado elementales si se desea implementar una escena de simulación compleja, siendo en este caso necesaria la utilización de una librería gráfica de alto nivel basada en la utilización de un *Grafo de Escena*. Los grafos de escena están diseñados para optimizar el proceso de dibujado y contribuir a la organización de la base de datos, resultando de especial importancia sus procesos de *culling* y selección de nivel de detalle, y su capacidad para actuar como soporte para la aplicación jerárquica de transformaciones afines. Se han presentado los tipos de nodos básicos empleados en la definición de estos grafos, y también la forma en la que estas librerías de alto nivel actúan en el caso de disponer de un sistema con varios procesadores.

Por último se ha realizado un recorrido por los grafos de escena actuales, profundizando en aquellos que han tenido una mayor influencia en la evolución de esta rama de la informática gráfica (OpenGl Performer, Open Inventor y VRML). El resto de los grafos de escena presentan unas características básicas similares, siendo destacable que alguno de ellos presentan carencias a la hora de proporcionar un buen rendimiento en tiempo real, que muchos de ellos reproducen el comportamiento de *OpenGl Performer*, y que algunos de ellos están siendo distribuidos como código abierto.

Del análisis del estado del arte, se ha determinado que *OpenGl* es la librería gráfica de bajo nivel que resulta más adecuada para actuar como base para todos los desarrollos realizados en este trabajo. Del análisis de las características de las librerías gráficas de alto nivel, se ha observado que la gran mayoría de las aplicaciones de simulación están desarrolladas en torno a un grafo de escena, y que éste constituye, por lo tanto, el substrato básico sobre el cual realizar las aportaciones que sean necesarias para el soporte de los actores virtuales. Los diferentes grafos de escena existentes presentan unas características básicas similares, de tal modo que uno de los objetivos de este trabajo es lograr desarrollos relacionados con actores virtuales que sean capaces de ser aplicados sobre cualquier grafo de escena que presente unos requisitos mínimos. OpenGl Performer será empleada como grafo de escena de referencia en aquellos casos en los que sea necesario mostrar de una forma concreta las relaciones entre las estructuras y métodos de gestión de actores virtuales, y el grafo de escena.

Capítulo 3. Actores Sintéticos en Tiempo Real: Estado del Arte.

3.1 Índice.

3.2 Introducción.

El objetivo de este trabajo es conseguir que los actores virtuales puedan ser integrados de una forma coherente con las estructuras y métodos existentes en la actualidad para creación de aplicaciones de simulación. Como se ha visto en el capítulo anterior, la informática gráfica en tiempo real se encuentra en un estado relativamente maduro, en el cual que la utilización de grafos de escena como *OpenGl Perfomer*, y librerías de bajo nivel como OpenGl constituyen una forma estándar de desarrollo de aplicaciones. En la actualidad, existen muchos aspectos relacionados con actores sintéticos en tiempo real que han sido estudiados con detalle, sin embargo, se puede decir que no existe un estándar para la definición de este tipo de actores, siendo común la tendencia a realizar desarrollos verticales, orientados a cubrir necesidades muy concretas del mercado, y a centrar las investigaciones en los aspectos de más alto nivel, sin profundizar demasiado en las estructuras de bajo nivel subyacentes, y dejando que requisitos tales como la velocidad o la integración en otras aplicaciones queden en un segundo término, por último, es también habitual que la mayoría de trabajos centren su atención en el desarrollo de actores de tipo humanoide.

El modelado de un actor virtual es una tarea muy compleja, que suele ser fragmentada en tres modelos diferentes: El ***modelo geométrico*** encargado de definir el aspecto externo del actor, el ***modelo motor***, encargado de controlar sus movimientos, y el ***modelo comportamental***, encargado de representar sus procesos mentales. Este capítulo comienza mostrando la forma en la que se realiza el modelado de los actores virtuales, y entrando a analizar el estado del arte actual de cada uno de los tres modelos. En un apartado final, se realiza un recorrido exhaustivo por los diferentes campos en los que en la actualidad se están aplicando este tipo actores virtuales, o bien de líneas de trabajo que encontrarán aplicación práctica en un futuro próximo.

3.3 Modelado de actores virtuales.

Para la creación de un actor virtual se ha de proporcionar un correcto modelo del cuerpo (estructura articulada, músculos, geometría que define la piel) y también un adecuado modelo de su cerebro, formado por centros de control del movimiento, percepción, comportamiento, aprendizaje, conocimiento, etc. Una de las primeras implementaciones completa de un actor virtual contemplando todos estos aspectos fue realizada por D.Terzopoulos [TER99]. En sus trabajos, la complejidad de la implementación de animales artificiales, fue abordada mediante una organización jerárquica en la que cada capa aumenta la funcionalidad de las anteriores (ver *Figura 3-1*). En la base se muestra la capa de modelado geométrico, que representa la morfología y apariencia del animal, sobre ella, las capas cinemática y física incorporan los principios biomecánicas que controlan la forma en la que dichos seres realizan sus movimientos. La capa comportamental permite que el actor sea capaz de realizar acciones en reacción las percepciones que tiene del entorno. En la parte superior se encuentra la capa del modelo cognitivo, que simula el comportamiento deliberativo de los animales superiores, almacenando el conocimiento que los animales tienen sobre si mismo y sobre su mundo.

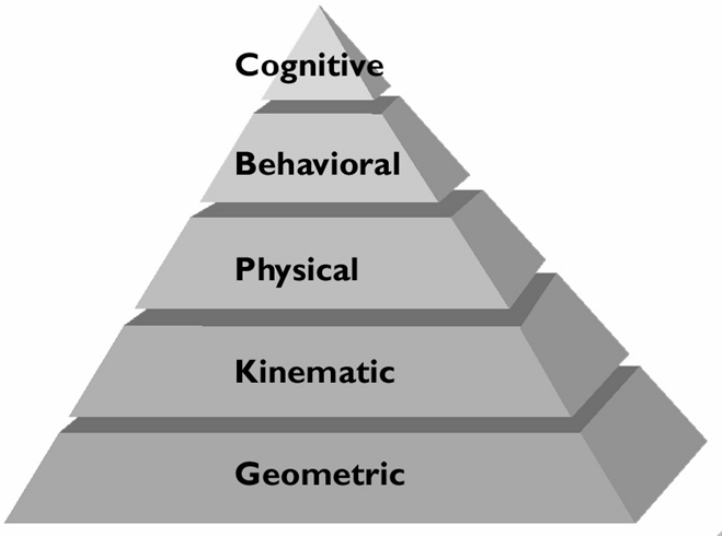


Figura 3-1. Capas necesarias en el modelado de un actor virtual según D.Terzopoulos.

Esta organización tan detallada es una réplica muy realista del sistema completo de un actor virtual, pero para realizar un estudio más cercano al estado del arte actual de los actores virtuales en tiempo real, se empleará una organización más abreviada (*Figura 3-2*):

- Modelo Comportamental. Representa los procesos mentales dentro del actor virtual englobando el sistema perceptivo, el sistema de aprendizaje, sus reglas de comportamiento y en los animales superiores, el sistema cognitivo. El control motor es también un proceso mental, pero por su intervención directa sobre la apariencia externa del actor virtual será incorporado dentro del modelo motor.
- Modelo Motor. Representa al centro motor del cerebro del animal, y a todas sus actuaciones sobre su sistema biomecánico, encaminadas realizar un determinado movimiento. Es un módulo físico-cinemático.

La entrada de datos son ordenes de alto nivel, la salida son valores que definen la configuración muscular del actor en un instante dado. Un modelo motor ha de considerar el centro motor del cerebro, el sistema biomecánico y una realimentación sensorial inconsciente (por ejemplo, detener un movimiento de alcanzar un objeto cuando se ha percibido una sensación de presión en los extremos de los dedos, cuando se ha oído el roce entre los dedos y el objeto, o directamente, cuando se ha visto que la mano ha alcanzado su objetivo). Entronca con la robótica por sus modelos dinámicos y cinemáticos complejos. Los parámetros de control enviados por el modelo comportamental al modelo motor son de alto nivel, del estilo "camina hasta aquella posición a una velocidad elevada". El modelo motor es el encargado de transformar estos parámetros de control de alto nivel, tales como la posición final y la velocidad de desplazamiento, en una secuencia detallada de actuaciones sobre su sistema biomecánico.

- Modelo Geométrico. En un animal real la entrada a este modelo sería una mezcla entre los impulsos eléctricos enviados a los músculos, y la respuesta del sistema biomecánico a estos impulsos. El objetivo final de los parámetros enviados por el modelo motor es conseguir que el actor adopte una determinada configuración esqueleto-muscular en un instante dado. En el caso de actores virtuales es común emplear una parametrización de la forma externa de un actor en función de los valores de los grados de libertad de sus articulaciones, y algún parámetro con información muscular complementaria (empleados por ejemplo para definir la expresión facial). La traducción entre representación paramétrica y la imagen final es un proceso que puede ser computacionalmente costoso formado internamente por varias etapas que serán posteriormente detalladas.

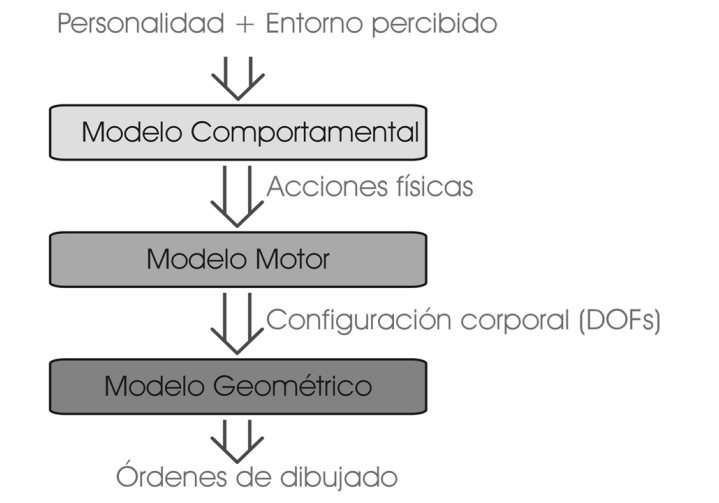


Figura 3-2. Flujo de información entre los distintos modelos que definen un actor virtual.

El estado del arte de cada uno de estos modelos será mostrado uno a uno en los siguientes apartados. El objetivo de este trabajo es encajar esta forma de organización lógica de un actor virtual con la forma de implementación de las aplicaciones actuales de simulación.

En la práctica, muchos de actores virtuales actuales no implementan los aspectos de más alto nivel, estando preparados para ser controlados de una forma más o menos directa por los usuarios de las aplicaciones. En este sentido, los actores virtuales pueden ser clasificados según su nivel de autonomía en las siguientes categorías:

- **De control directo.** El actor virtual es controlado directamente a través de distintos tipos de sensores conectados al usuario o usuarios del sistema. El actor virtual no tiene ningún tipo de comportamiento inteligente, una o varias personas definen en tiempo real su posición de una forma similar a como se haría con una marioneta. Es usual que las personas que controlan al actor virtual utilicen algún tipo de periférico especial como guantes de datos o sistemas de seguimiento de posición (serán detallados en el apartado 3.5.2). Este tipo de control es empleado en sistemas de realidad virtual inmersiva que empleen casco de realidad virtual y necesitan representar al cuerpo del usuario, así como para controlar los presentadores virtuales empleados en la industria televisiva.

- **Semi-autónomos.** En este caso, las actividades principales del actor son dirigidas por un usuario, bien sea por medio de menús, selección de alguna combinación de botones, la utilización del ratón, un guante de datos o un joystick. Un ejemplo claro de control semi-autónomo sería el de un actor al que se le indica mediante el ratón del ordenador el punto 3D al que tiene que desplazarse, en este caso, las ordenes de movimiento provienen del usuario, pero el actor dispone de un modelo motor capaz, es este el que ha de calcular la trayectoria, y realizar de forma autónoma los movimientos necesarios para desplazarse hasta ese punto. Este tipo de actores también pueden ser controlados mediante interfaces basadas en lenguaje, consistentes en la utilización de frases imperativas que han de estar dentro de un conjunto de expresiones conocidas por el actor **[PER95] [STA97] [SMI97]**. Estos actores han de tener motores de comportamiento suficientemente genéricos, y, además, analizar el contexto en el que se encuentran para poder interpretar correctamente las ordenes.

- **Autónomos.** El caso de mayor sofisticación sería el de aquellos actores que disponen de un control totalmente autónomo, que hace que se comporten como si estuviesen dotados de vida propia, e incluso de inteligencia, siendo capaces de percibir el entorno en el que se encuentran y actuar en función sus motivaciones internas. Conseguir que un actor sintético, sea capaz de reaccionar de una forma natural en función de los estímulos del entorno en el que se encuentra, se mueve en un nivel de complejidad totalmente distinto a lo que es la simple aplicación de valores pregrabados (provenientes por ejemplo de captura de movimiento) sobre una estructura articulada. El actor ha de tener un nivel de conocimientos e inteligencia suficiente para adaptarse a su entorno físico, realizar determinadas tareas dentro de dicho entorno, mostrar una determinada personalidad y ser capaz incluso de realizar trabajos colaborativos con otros actores o incluso con personas reales **[MAE95] [ROU96].**

En algunas publicaciones es habitual denominar a los actores con control directo o semiautónomo *avatares* y denominar *agentes* a los actores que disponen de un control autónomo.

3.4 Modelo Comportamental.

Un actor virtual ideal debe ser capaz de mostrar un comportamiento de una complejidad similar al de sus homólogos en el mundo real, es decir, ha de ser capaz de reproducir distintos procesos mentales tales como el sistema perceptivo, el sistema de coordinación de movimientos, el sistema de aprendizaje, sus reglas de comportamiento, y en los animales superiores, el sistema cognitivo **[TER99]**. Ésta es un área enormemente amplia, íntimamente ligada con la inteligencia artificial y que está centrando y ha centrado la atención de la mayoría de investigaciones sobre actores virtuales.

El objetivo principal del modelo comportamentel es conseguir actores con un control autónomo. Es habitual modelar el comportamiento de un actor virtual como una serie de mecanismos de control que actúan de forma paralela. Estos mecanismos de control suelen integrar la información proporcionada por el sistema perceptivo y cognitivo, y ordenar algún tipo de actuación sobre el sistema motor. Para ilustrar el nivel de dificultad que puede tener la implementación de algunos mecanismos de control autónomo se va a mostrar cuales serían las estrategias que se podrían emplear para implementar un mecanismo de control autónomo capaz de realizar la tarea, aparentemente sencilla, de alcanzar y agarrar un objeto **[DOU96]**. El método ideal para conseguir este objetivo sería hacer que los actores fuesen capaces de identificar los objetos, determinar su utilidad y agarrarlos empleando su experiencia. En la actualidad, aún estamos lejos de utilizar este tipo de soluciones. Una alternativa sería diseñar un sistema experto con recopilación exhaustiva de todos los tipos de objetos y todas las formas posibles de agarrarlos, y, además, dotar a los actores sistema de reconocimiento de patrones que idendificase la forma de un objeto concreto con alguno de los contemplados por el sistema experto. Los dos métodos anteriores constituirían una forma adecuada de definir este mecanismo, sin embargo, la excesiva complejidad del problema hace que se tenga que recurrir a soluciones más acordes con el estado de la ciencia en áreas como la visión e inteligencia artificial. Una solución práctica **[LEV96]**, consiste en hacer que cada objeto susceptible de ser agarrado disponga de una tabla con información asociada sobre cual es la forma adecuada para agarrarlo. La mano ha de cerrarse sobre el objeto basándose en la información geométrica del objeto y detectando las posibles colisiones. El proceso puede ser aún más complicado si se quiere que la geometría de las yemas de los dedos se modifique debido a la presión ejercida sobre el objeto **[GOU89].**

Entre los mecanismos de control autónomo de los actores virtuales tienen especial importancia el *control de la atención* y el *control de la locomoción*. El control de la atención es llevado a cabo por motores de comportamiento que se encargan de tareas tales como hacer que los ojos y la cabeza de un actor sigan un punto de interés determinado **[CHO95][CHO99]**. El control de locomoción es el encargado de modificar la posición del actor sobre el mundo virtual, serían ejemplos de controles de este tipo **[BAD96] [ASH01],** uno que gobernase el desplazamiento de un actor por una escena con el propósito de alcanzar algún objetivo, uno que hiciese que el actor siempre se estuviese escondiendo del resto de actores existentes en la escena, uno que gestionase la búsqueda de un objeto en la escena, o uno de persecución, en la que el objetivo cambia dinámicamente y el actor tiene que planificar continuamente nuevas trayectorias.

En el caso de actores humanos también resultan fundamentales los mecanismos de *control de comunicación no verbal* **[CAS94][BAT94]**, los cuales acompañan a las locuciones, haciendo expresiones con las manos y el cuerpo que reflejan el estado de animo del personaje, enfatizando su discurso.

También existen *controles cooperativos* que se encargarían de la coordinación de múltiples actores en la realización de tareas de forma cooperativa. Serían ejemplos de este tipo de tareas el hecho de coger una caja entre dos de ellos, o de saludarse estrechando la mano. Este tipo de controles cooperativos suponen la primera etapa de los que serían los mecanismos necesarios para gestionar una sociedad virtual.

Para ilustrar la forma en la que varios controles se pueden estar ejecutando de forma paralela, observemos el ejemplo de un actor humano que necesita acercarse a un lugar y agarrar un determinado objeto. Sería necesario que el actor se desplazase hasta la zona en la que este el objeto haciendo que los pies sigan la trayectoria adecuada **[BEC97][KOG94][REI94]**, simultáneamente, mecanismos de control de atención mostrarían como el actor está buscando con la mirada la posición del objeto. También sería necesario un mecanismo de control encargado de evitar obstáculos que puedan estar cerca de los pies, así como analizar la posible aparición de objetos móviles que interfieran la trayectoria, por último sería necesario utilizar controles adecuados para alcanzar y agarrar el objeto. De forma paralela a estas actividades, otros controles podrían estar gestionando ciertas actividades automáticas realizadas por el actor, tales como el parpadeo y la respiración o incluso actividades más complejas tales como mantener una conversación. También es necesario que existan métodos que prioricen las actuaciones en caso de que se produzcan conflictos, por ejemplo los gestos manuales derivados de la expresión, con otro tipo de actividad manual como agarrar y manejar algún objeto.

Existen arquitecturas especiales capaces de realizar una descripción de aloto nivel de los comportamientos y sus interrelaciones mediante máquinas de estados como el caso *PaT-Nets* (*Parallel Transition Networks*) **[BAD93][BAD98]** empleadas en la gestión del actor sintético *Jack*, o la librería *AGENTlib*, desarrollada en el marco del proyecto *VRML Humanoid* **[BOU97]** consistente en la gestión de múltiples procesos paralelos a los que llaman acciones, y la transición suave entre dichas acciones.

La gestión del comportamiento de los actores virtuales es un área de investigación enormemente atrayente y compleja, que conecta directamente con áreas del conocimiento tales como la inteligencia artificial, la etología, psicología o la sociología. En los últimos años han sido llevadas a cabo multitud de investigaciones en esta área, orientadas desde muy distintos enfoques: sistemas basados en una estrategia de percepción-acción **[REY87][MCK90]**, sistemas basados en agentes inteligentes **[MAH94][CREM96]**, sistemas reactivos **[FEI96]**, sistemas basados en reglas **[CAS94][BEA95]**, sistemas basados en redes neuronales **[BEE90] [ISL01]** , basados en el aprendizaje **[BLU02][YOO00]**, mecanismos de acción-selección **[PIN00][MAE94][TU94][NOS96] [THA96] [BLU97]**, basados en un modelo cognitivo **[TER99][FUN99][FUN00]**, sistemas basados en guiones **[PER96] [GOL97]**, modelado de la personalidad y las emociones **[RIC99][PET97][BAD97][HAY97] [BAT94]**. Para una revisión más detallada se puede consultar **[BADa02]**.

3.5 Modelo Motor.

El centro de gestión del comportamiento envía al sistema motor ordenes relativas a la realización de movimientos. La comunicación entre el cerebro de un animal real y su sistema motor consiste básicamente en la transmisión de determinados impulsos eléctricos a los músculos, los cuales reaccionan adoptando la postura ordenada desde el cerebro. En el caso de los actores virtuales la barrera entre el modelo motor y comportamental no es tan clara (especialmente en aquellos casos que impliquen movimientos complejos).

Por modularidad resulta adecuado que el modelo de gestión de comportamiento envíe ordenes de alto nivel del estilo "mira hacia ese punto", "camina en esta dirección", "muestra una expresión facial sonriente" o "parpadea de una forma más rápida", al modelo motor. El modelo motor del actor virtual sería responsable de traducir todas estas ordenes en una secuencia de actuaciones sobre su estructura ósea y muscular.

Algunos tipos de movimientos tales como el parpadeo, o la secuenciación de pasos durante la locomoción, pueden ser realizados de una más o menos automática, pero en algunos casos existen ordenes relacionadas directamente con la animación del personaje (ver el ejemplo de alcanzar y agarrar un objeto presentado en el apartado anterior) que implican a los centros de percepción, cognitivo y de aprendizaje de modelo comportamental. En estos casos, el modelo comportamental ha de realizar las operaciones necesarias para reducir la complejidad de las ordenes enviadas al sistema motor.

Cualquier pequeño desajuste en la forma en la que un actor virtual realiza sus movimientos resulta fácilmente perceptible a simple vista. Ésta es una de las razones por la cual la implementación del modelo motor es una de las tareas más difíciles en el proceso global de creación de un actor virtual, y la razón que justifica que la utilización de movimientos grabados mediante técnicas captura de movimiento sean las más empleadas en la actualidad, incluso en sistemas que no presentan las limitaciones del tiempo real (por ejemplo en la industria cinematográfica).

En los siguientes apartados, se detallan las distintas técnicas empleadas en la actualidad en la implementación del modelo motor de los actores virtuales, realizando una comparación entre las ventajas en inconvenientes de la utilización de movimientos grabados o la síntesis de movimiento, mostrando los actuales sistemas de captura de movimiento, empleados tanto para la generación de secuencias de movimiento como para gobernar los actores de control directo, y por último, mostrando un estado del arte de las técnicas de control de la animación facial de actores humanos.

3.5.1 Movimientos Grabados vs. Síntesis de Movimiento.

La forma más adecuada de control del movimiento de un actor virtual sería la síntesis de movimiento, sin embargo, la dificultad actual para conseguir movimientos realistas, estables y compatibles con el tiempo real,

hace que las técnicas basadas en la utilización de movimientos generados en una fase previa a la simulación sean muy habituales.

Los datos de movimientos grabados pueden haber sido generados mediante *keyframing* en un programa de animación 3D tradicional (es decir, programas que permiten definir la postura de un actor en distintos instantes del tiempo y realizar una interpolación entre estas posturas), o bien, pueden haber sido obtenidos mediante algún sistema de captura de movimiento (*motion capture* o *mocap* en literatura inglesa). La ventaja principal de los movimientos grabados es que su utilización es tan simple como aplicar directamente los datos existentes en una tabla sobre las articulaciones, resultando, además muy rápido computacionalmente. Su principal inconveniente es su poca variabilidad (todos los actores se mueven de un modo similar), y su poca reusabilidad (la modificación de las distancias entre articulaciones hace que los movimientos dejen de ser adecuados). Otro inconveniente se produce en el caso de que sea necesaria realizar una transmisión por red, puesto que sería necesario transmitir todos los valores de sus grados de libertad varias veces por segundo. Los movimientos grabados han de ser aplicados en un actor que presente las mismas características antropométricas que el actor a partir del que se ha hecho la captura de los movimientos, y tan sólo es posible modificar la orientación global del actor o modificar la velocidad con la que se ejecuta la secuencia. Para enriquecer la utilización de este tipo de técnicas se pueden realizar ediciones sobre el movimiento grabado, añadiendo por ejemplo ruido periódico a los valores de los grados de libertad **[PER96].** Trabajos recientes proponen soluciones al problema de la limitación antropométrica de diferentes formas **[BRU96][HOD97][GLE98][LEE99]**.

La síntesis de movimiento utiliza distintos tipos de técnicas para calcular los valores angulares de las articulaciones en función de métodos de control de alto nivel: cinemática inversa, generadores de locomoción, generadores de expresión facial, métodos para coger/manipular objetos, controladores de equilibrio, etc. Las ventajas principales de la síntesis de movimiento son su reusabilidad, y su control de alto nivel, esto último permite la definición de movimientos complejos a partir de muy pocos parámetros. Esta particularidad presenta como ventaja derivada que pueden facilitar enormemente la transmisión por red. Su principal desventaja reside en la dificultad para conseguir movimientos con un aspecto realista (la *captura de movimiento* es el método que actualmente presenta mejores resultados). Los cálculos necesarios pueden ser muy costosos para la CPU, e incluso es bastante habitual que los algoritmos de control produzcan indeterminaciones o resultados imprecisos que den como resultado posturas incorrectas. En la actualidad existen varias lineas de investigación centradas en este area que están proporcionando resultados muy interesantes **[FAL01] [KIM03] [LAS00] [LI02] [KOV02] [TOL00] [BAD99] [FAN03].**

Una alternativa práctica a la utilización de movimientos sintéticos consiste emplear movimientos obtenidos mediante captura de movimiento, y segmentarlos en bloques que pueden ser nombrados, almacenados, y posteriormente ejecutados por separado **[GRA95]**, siendo posible la interconexión de varios movimientos mediante diferentes tipos de interpolación **[BRU95][ROS96][POP99],** así como la síntesis de nuevos movimientos basados en los datos capturados **[ARI02][LIU02][DAS99][TAN00][LEE02]**. Este tipo de técnicas es ampliamente utilizado en el mundo de los videojuegos.

3.5.2 Sistemas de captura de la posición corporal.

Una de los campos relacionados con los actores virtuales hacia el que se han orientado más esfuerzos ha sido el desarrollo de sistemas de seguimiento de la posición corporal. Este campo ha tomado especial importancia por dos motivos principales: por un lado constituyen la base para conseguir las secuencias de movimiento grabado citadas en el apartado anterior, y por otro lado son totalmente necesarios para controlar al actor virtual que representa al usuario en caso de que esté empleando un casco de realidad virtual y desee ver una representación de su cuerpo. Existen sistemas que están centrados en la captura de la posición del cuerpo, otros que capturan la posición de las manos, y otros que capturan la expresión facial.

Dentro de los sistemas que capturan la posición corporal del actor se puede hacer una clasificación atendiendo al tipo de tecnología que emplean:

- **Mecánicos.** La persona real viste una estructura exoesqueletica formada por un conjunto de barras de metal unidas mediante un tipo especial de juntas que llevan codificadores angulares. Todo el conjunto es fijado a la espalda mediante un dispositivo en forma de mochila. Cuando la persona se mueve fuerza a que dicho exoesqueleto también se modifique y esto hace posible conocer la postura del actor analizando los valores suministrados por los sensores. Existen versiones de este tipo de sistemas que están especializados en analizar la posición de la mano, del brazo o incluso de la cara. Existen también versiones de este tipo de aparatos que no se colocan sobre directamente sobre el cuerpo, sino que actúan como periféricos para realizar *keyframing* (un ejemplo de esto es el *Monkey*, un periférico desarrollado por Digital Image Design que es pequeño maniquí articulado formado por 39 juntas). Estos sistemas presentan como principal ventaja el hecho de no sufrir ningún tipo de interferencia. Por contra, es un sistema que necesita ser recalibrado cada vez que se viste, es un sistema local (no proporciona valores absolutos), y puede resultar incomodo. Es un tipo de sistema al que se le puede añadir realimentación de fuerza.

- **Ópticos.** El usuario viste un traje que lleva adherido un conjunto de pequeñas esferas de material reflectivo y se mueve dentro de una habitación que está rodeada de múltiples cámaras colocadas en posiciones conocidas. La posición espacial de cada una de las esferas es obtenida por triangulación. Es cómodo porque el usuario, aunque ha de llevar un conjunto de 20 o 30 bolas adheridas a su cuerpo, no necesita ningún tipo de aparataje electrónico o mecánico sobre su cuerpo, y es posible definir grandes zonas de movimiento. Proporciona valores absolutos, es posible hacer que se capturen movimientos de varios actores simultáneamente, y sus datos contienen muy poco ruido. Un inconveniente es que necesita una habitación especial en la que exista una gran cantidad de cámaras para evitar los problemas de oclusión, esto hace que además resulte un sistema más caro que los otros.

- **Electromagnéticos.** El usuario lleva adheridos a su cuerpo un conjunto de receptores electromagnéticos que son capaces de conocer su posición y orientación respecto a una base fija que hace de emisor. Las posiciones y rotaciones son proporcionadas en valores absolutos. Pierde precisión rápidamente a medida que el usuario se aleja de la base emisora, y sufren interferencias con elementos metálicos.

Para la captura de la expresión facial existen también sistemas mecánicos, que consisten en un casco del que salen una serie de sensores de presión que entran en contacto con distintos puntos de la piel de la cara. También existen distintos tipos de sistemas de captura de la expresión facial mediante procesamiento de imagen, algunos de ellos analizan directamente la expresión facial mediante algoritmos de detección de puntos relevantes de la cara tales como la comisura de los ojos, los párpados etc. y otros más sencillos determinan la posición de unos adhesivos especiales que han sido previamente fijados sobre la cara.

Para la captura de la postura de las manos un tipo especial de periférico en forma de guante (denominados bajo en nombre genérico de guantes de datos o *dataglobes* en literatura inglesa) que utilizan muy variadas técnicas de sensorización para conocer la posición de sus articulaciones: mecánicas, piezoeléctricas, ópticas...

3.5.3 Animación Facial.

Como ya se ha comentado anteriormente, la mayoría de trabajos relacionados con actores virtuales han estado centrados en la definición y gestión de actores de tipo humanoide, y dentro de éstos, uno de los aspectos más tratados ha sido el de la comunicación facial, de hecho, existen cierto tipo de aplicaciones (videoconferencia, presentadores virtuales, cabezas parlantes) que centran su atención únicamente en este aspecto.

La expresión facial tiene un papel básico en la comunicación, es capaz de expresar el estado emocional, así como de enriquecer la comprensión de una determinada locución, además el movimiento de los labios desempeña un papel muy importante en el entendimiento de una conversación, especialmente en entornos en los que las condiciones acústicas no son ideales.

Existen cuatro métodos básicos **[CAP97]** que pueden ser utilizados para controlar la expresión facial de un actor virtual:

- Texturado de la cara mediante una secuencia contínua de vídeo. El usuario ha de estar físicamente delante de una cámara de vídeo, en una posición tal que ésta capture su cabeza y sus hombros. Es necesario emplear un algoritmo de reconocimiento de imagen que sea capaz de determinar la posición de la cabeza en cada instante, de este modo, el trozo de imagen que se corresponde con la cara del usuario puede ser mapeada sobre una cabeza virtual. Este tipo de técnica puede ser empleada para el control de avatares, videoconferencia o para entornos colaborativos.

- Codificación de las expresiones faciales basadas en un modelo. Al igual que el caso anterior, el usuario ha de estar delante de una cámara de vídeo que captura imágenes de la cara y hombros de una persona real. El método de procesamiento de imágenes es mucho más complejo que el anterior y que extrae parámetros tales como los valores de azimut, elevación y twist de la cabeza, así como la apertura de la boca, la elevación o distancia entre las cejas, y la posición horizontal y vertical del iris. Los parámetros de expresión facial obtenidos de este procesamiento pueden ser almacenados empleando distintos tipos de codificación: El *Facial Action Coding System* (FACS) que proporciona un conjunto de *Action Units*

(AU) los cuales representan la actuación de determinados grupos musculares que actúan juntos para producir un determinado movimiento simple **[EKM78]**, la utilización del conjunto básico de acciones faciales (conocidas como MPAs o *Minimal Perceptible Actions*) descritas en **[KAL93],** el modelo parametrizado propuesto en **[PAR82]** o los FAPs (*Facial Animation Parameters*) propuestos por el estándar MPEG-4 **[LAV99]**. La codificación de la expresión facial en función de este tipo de parámetros resulta un método ideal para realizar transmisiones por red.

- Generación automática del movimiento de los labios a partir del habla. Una forma alternativa de definir la expresión facial sin necesidad de emplear un método que analice directamente la expresión de la cara, consiste en utilizar métodos de reconocimiento del habla, y a partir de ahí, extraer las correspondientes posiciones de la boca.

- Expresiones predefinidas. Consiste en tener almacenada una lista con las expresiones faciales más comunes (alegría, sorpresa...), que pueden ser asignadas al actor de forma muy sencilla, bien sea mediante la pulsación de una tecla, o conjuntos de caracteres similares a los "smileys" utilizados en correo electrónico.

3.6 Modelo Geométrico.

Los valores de salida del modelo motor son una serie de parámetros de control mediante los cuales se define la postura del actor, dichos parámetros hacen referencia a actuaciones sobre los grados de libertad de las articulaciones (dofs), y a veces, sobre los músculos (expresión facial y otros). El modelo geométrico de un actor virtual en tiempo real, es responsable de transformar esta representación parametrizada, en un conjunto de primitivas de dibujado (representando la piel del actor), que serán enviadas al hardware gráfico.

Esta transformación suele ser realizada mediante dos pasos intermedios (*Figura 3-3*), en el primero de ellos, se hace que el esqueleto del actor adopte la forma indicada por los parámetros de postura, y en el segundo se hace que la forma de la piel se adapte a la transformación sufrida por el esqueleto.

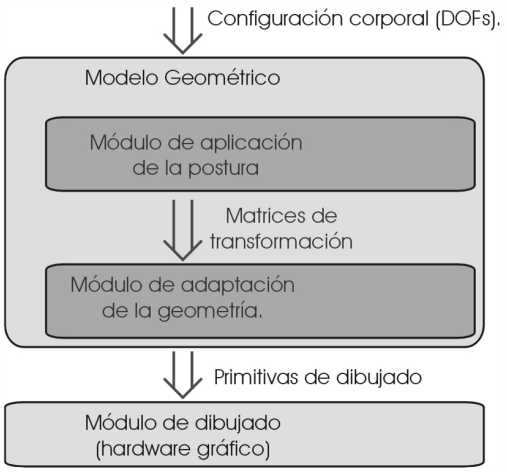


Figura 3-3. Etapas que componen el Modelo Geométrico de un actor virtual.

El módulo de aplicación de la postura, calculará y combinará las matrices de transformación de cada punto de articulación empleando para ello la información topológica del actor. Con ello consigue que el esqueleto del actor adopte la postura final deseada. El módulo adaptador de la geometría consiste en algún método o conjunto de métodos, que hacen que la nueva postura del esqueleto del actor afecte a los vértices que definen su geometría externa. Todas las etapas previas al módulo de dibujado acaban transformando al actor en una estructura geometría estática, formada por un conjunto de triángulos y otras primitivas de dibujado. El hardware gráfico transformará estas primitivas en la representación bidimensional del actor virtual.

La forma en la que se realiza la parametrización de la postura de los actores virtuales es de suma importancia, ya que constituye un aspecto básico para poder definir un estándar de actores. Este tipo de estandarización hasta la actualidad solamente se ha dado en actores de tipo humanoide, y afecta no solamente a parámetros de control sino también a la forma topológica. El hecho de que la postura de un actor pueda ser controlada mediante un conjunto reducido de parámetros es también de vital importancia en la realización de simulaciones distribuidas.

En los siguientes apartados se presentan las técnicas empleadas en la actualidad para la representación de la apariencia externa de los actores virtuales, así como los trabajos relacionados con la estandarización en la parametrización de actores de tipo humanoide y la forma en la que este tipo de parametrización está siendo usada para facilitar la realización de simulaciones distribuidas en las que intervengan actores virtuales.

3.6.1 Técnicas de adaptación y representación en Tiempo Real de la geometría.

En la actualidad existen varios métodos de definición de la geometría de personajes sintéticos tridimensionales. En animación 3D tradicional (no tiempo real) se pueden emplear técnicas muy sofisticadas para representar la geometría de los actores y sus deformaciones: Superficies equipotenciales (metaballs), superficies paramétricas (NURBS y otras), técnicas de deformación del espacio (diversos tipos de FFDs), o modelos multicapa basados en la simulación de los músculos y las distintas capas de tejido. Todas estas técnicas consiguen que el actor pueda ser representado con una piel contínua que recubre su cuerpo (*skining* en literatura inglesa), obteniendo actores sintéticos con un nivel de acabado tan elevado que los hace casi indistinguibles de sus homólogos reales, de hecho, en la industria cinematográfica comienza a ser muy habitual la utilización de animales virtuales en aquellos casos en los que los reales estén extinguidos (por ejemplo los dinosaurios), pueden ser peligrosos (por ejemplo boas, tiburones, tigres o rinocerontes), o sean difíciles de filmar (por ejemplo pulgas). Sin embargo, estos métodos son difíciles de implementar haciendo que cumplan con los requisitos necesarios un sistema de gráficos en tiempo real. Los actores en tiempo real actuales presentan unos modelos de representación geométrica mucho más sencillos, sin embargo existe una tendencia natural a conseguir que su acabado sea cada vez más realista, y en casos muy puntuales se han desarrollado actores virtuales en tiempo real que tienen un nivel de acabado comparable a los empleados por la industria cinematográfica, si bien, es cierto que es a costa de haber hecho un desarrollo específico y de consumir muchos recursos (por ejemplo, la animación de un único presentador virtual para televisión, suele requerir la utilización de una Sgi Onyx Infinity Reality con varias CPUs).

En los siguientes apartados, se describen las técnicas empleadas para la adaptación y representación de la geometría externa de los actores virtuales en sistemas en tiempo real. Los métodos más habituales consisten en el empleo de geometrías rígidas interconectadas, y en la técnica de *clustering* de vértices. En un último apartado se describen modelos más complejos susceptibles de ser empleados en la representación de actores virtuales.

3.6.1.1 Geometrías rígidas interconectadas.

Una solución muy sencilla y eficiente, consiste en definir el cuerpo del actor como un conjunto de elementos rígidos interconectados, el nivel de acabado puede ser aceptable si se hace que los puntos de interconexión entre las geometrías tengan un acabado esférico, y se hace una buena utilización de las texturas. El empleo de elementos rígidos reduce considerablemente el coste computacional de la gestión de un actor virtual, y hace que la definición de varios niveles de detalle para el actor resulte muy sencilla.

3.6.1.2 Clústering de vértices.

La utilización de elementos rígidos interconectados tiene como principal inconveniente que la sensación de continuidad en la piel resulta a veces insuficiente (especialmente en modelos que necesiten animación facial, o que tengan muy pocos vértices). En estos casos es posible utilizar la técnica denominada "*clústering de vértices"*, consistente en la elaboración de modelos geométricos con muy pocos vértices, y hacer que su posición se vea afectada por los valores de los grados de libertad de las articulaciones. Este tipo de técnica es ampliamente utilizada en el desarrollo de videojuegos. La desventaja principal del *clústering de vértices* es que resulta muy poco reutilizable, dado que para proporcionar buenos resultados es necesario personalizar la respuesta de cada vértice o grupo de vértices con respecto a las modificaciones de los grados de libertad de los que dependen, personalización que es realizada en la mayoría de los casos a nivel del lenguaje de programación. Por otro lado, la gestión de niveles de detalle no es en la actualidad considerada, y resulta de difícil implementación.

3.6.1.3 Modelos complejos.

Tanto el modelo de geometrías rígidas interconectadas como la utilización de clústering de vértices presentan claras limitaciones en cuanto al nivel de calidad visual que puede ser obtenido. Con el fin de conseguir mejores resultados se está experimentando con la utilización de modelos más complejos que ya han sido utilizados en animación 3D tradicional. En este apartado se realiza un recorrido por los métodos más avanzados de representación de la geometría de los actores virtuales.

Las técnicas de deformación del espacio, consideran que los vértices de un determinado objeto están inmersos dentro de un determinado volumen definido a partir de varios puntos de control. La modificación de la posición de esos puntos produce una deformación sobre el espacio que definen, y, en consecuencia, sobre los vértices que están en su interior. La técnica de deformación del espacio más utilizada es son las *Free Form Deformations* (FFD) **[SED86]**, que puede ser empleada en actores virtuales pare definir una capa intermedia entre la piel y el hueso **[CHAD89].** El uso estandarizado de este tipo de técnicas proporciona buenos resultados, pero su control no resulta sencillo. En la actualidad existen distintos tipos de *FFDs* como las *extended FFDs* **[COQ90]**, las *FFDs de manipulación directa* **[HSU92]**, las *FFDs basadas en Nurbs* **[LAM94]**, *FFDs de topología arbitraria* **[MAC96]**, *Rational FFDs* **[KAL92]** y *Dirichlet FFDs* **[MOC96]**. Las *FFDs* han sido utilizadas para simular la capa muscular que controla la expresión facial **[KAL92]**, también para controlar la animación de articulaciones sencillas **[LAM94],** y en la animación de estructuras articuladas complejas tales como una mano una mano **[MOC96].** Las técnicas de deformación del espacio, permiten deformar directamente los vértices de una primitiva de tiempo real (por ejemplo una triangle strip), y pueden ser computacionalmente poco costosas.

Mediante la utilización de superficies paramétricas es posible definir una superficie tridimensional suave a partir de unos pocos puntos de control **[CAT78] [DER98]**. Para su utilización en actores virtuales, se establece una relación entre los puntos de control de la superficie, los huesos y los grados de libertad del actor, siguiendo una filosofía similar a la empleada en la técnica de clústering de vértices. Presenta unos resultados muy buenos, pero necesita que el modelo geométrico haya sido totalmente construido a partir de superficies paramétricas mediante algún tipo de herramienta especial.

Si bien los dos modelos anteriores son los más comúnmente empleados, presentan ciertas limitaciones intrínsecas que están intentando ser superadas mediante el empleo de estructura multicapa (hueso-músculo-piel) **[BERT97] [CHAD89] [MON91] [SCH97].** En los últimos años ha surgido una línea de investigación centrada en la definición de modelos de piel animable a partir de los datos proporcionados por algún escáner tridimensional **[ALL02] [ALL03] [MOH03],** o extraídos de secuencias de vídeo **[BRA01] [SAN03].** Así mismo, se ha investigado sobre nuevos métodos de animación que permiten controlar la deformación de las superficies de piel definidas mediante estas técnicas **[WAN02] [LEW00][CAP02]**.

En el caso de humanoides virtuales es bastante habitual hacer una distinción entre la animación del cuerpo y la animación facial, e incluso la animación de las manos suele ser tratada por separado. Algunas aproximaciones emplean modelos de deformación del espacio, o incluso geometrías rígidas para definir el cuerpo, y animan la cara empleando otro tipo de técnicas. El modelo para definir la piel de la cara humana es más complicado que el del cuerpo, puesto que viene definida básicamente por la acción de los músculos faciales, y no por la posición de los huesos (a excepción de la mandíbula). La mayor parte de la capacidad expresiva del actor se centra en la cara, y por tanto, cualquier pequeña incorrección resulta fácilmente perceptible. Los modelos empleados en su representación suelen utilizar métodos multicapa hueso-músculo-piel **[KAL91]**, que tienen en cuenta los diferentes tipos de músculos que actúan sobre la expresión, las distintas capas tejido existentes debajo de la epidermis, y su forma de reaccionar a las fuerzas a las que se ven sometidos **[TER90][WAT87]**, contemplándose incluso los efectos de la edad sobre la piel **[VIA92][WU96]**, así como modelos basados en morphing geométrico **[BLA99].** Las manos son también un elemento imprescindible en la comunicación no verbal de los humanos y existen líneas de investigación centrados específicamente en esta área **[MAG88][GOU89][UDA93][DEL93] [MOC96].** De igual modo también se han hecho trabajos centrados en la gestión de la ropa utilizada por los actores virtuales **[NG96][BAR98][VOL00]**, así como de su pelo **[MAG00][CHA02][KIM02]**,

La capacidad del hardware gráfico actual de poder ejecutar código que modifique los vértices de la geometría (vertex shaders), abre la puerta a la implementación de sistemas de skining de la geometría muy rápidos.

En general, se puede decir que sigue siendo necesaria obtención de modelos de representación geométrica que sean suficientemente genéricos, y presenten una buena calidad gráfica al mismo tiempo sean computacionalmente poco costosos. Dichos modelos han de ser capaces de llevar una adecuada la gestión de niveles de detalle, y también han de ser capaces de gestionar de forma adecuada y en tiempo real aspectos como son la ropa, el pelo o los pliegues que se producen en la piel, el plumaje o escamas.

3.6.2 Trabajos de estandarización en actores humanoides.

Como ya se ha indicado, existe una tendencia a que los trabajos realizados en áreas relacionadas con actores virtuales tengan una estructura vertical. Una de las pocas excepciones a esta tendencia son las propuestas de estandarización de actores virtuales humanos para su utilización en Internet. Estas propuestas se basan en la ampliación de estándares ya establecidos como son el MPEG y VRML.

El estándar MPEG-4, por un lado define una serie de parámetros antropométricos básicos, a partir de los cuales es posible modificar la geometría de un actor humano neutro adaptándola a la morfología de una persona concreta, y por otro lado, un conjunto de parámetros a partir de los cuales es posible controlar su animación corporal y facial. Para el estándar VRML, se describe un método para crear actores virtuales humanos a partir de nodos VRML especializados, así como la nomenclatura empleada para dichos nodos y su organización topológica. Estas dos propuestas son analizadas con mayor profundidad en los siguientes apartados.

3.6.2.1 Animación de humanos en MPEG-4 .

El estandar visual MPEG-4 **[KOE97]**, permite la codificación simultanea de datos sintéticos y naturales, de audio y vídeo, en sistemas que tienen limitado el ancho de banda y capacidad de almacenamiento, prestando atención a la escalabilidad y sincronización en tiempo real **[DOE97]**. Resulta especialmente interesante su capacidad de codificar, conjuntamente, imágenes y vídeo naturales (basadas en pixeles) junto con parámetros descriptivos de geometría sintética 2D o 3D (por ejemplo textos vectoriales, o una representación 3D de un videoconferenciante).

Uno de los aspectos que centran más atención en este formato es la codificación de animación facial y corporal de personajes 3D de tipo humanoide, así como, su sincronización con voz natural, o sintetizada a partir de texto **[SNHa96][SNHb96]**. MPEG-4 no estandariza los modelos geométricos que representan la cara o el cuerpo, pero si los parámetros a partir de los cuales una cara o cuerpo neutral (también especificados por el estándar), pueden ser personalizados y animados en tiempo real. El formato MPEG-4 controla por separado la animación facial y la corporal. Propone como estándares los parámetros antropométricos de definición de la cara y el cuerpo (**FDP** y **BDP**), y también los parámetros de que definen la expresión facial y la postura de cuerpo (**FAP** y **BAP**).

MPEG-4 define un conjunto de 84 puntos tridimensionales destinados a parametrizar la forma de la cara sintética también llamados FDPs (*Facial Definition Parameters*). La parte del estándar encargada de controlar la animación facial proporciona un conjunto de 68 parámetros a los que llama FAPs (Facial Animation Parameters), 66 de estos parámetros proporcionan información de bajo nivel (como por ejemplo podría ser el desplazamiento vertical del párpado del ojo izquierdo) y existen 2 FAPs de alto nivel, que son utilizados para representar con un solo parámetro, la mayoría de las expresiones faciales comunes (alegría, tristeza, enfado, miedo,...), y los visemas (las posturas de la boca correspondientes a los distintos fonemas).

Tanto los FAPs como los FDPs están medidos respecto a una cabeza neutral cuyas características también son recogidas por el estándar **[LAV99]**, los desplazamientos durante la animación se miden en unas unidades específicas llamadas FAPU (Facial Animation Parameter Units) que representan fracciones de medidas características de la cara. Muchos de los trabajos actuales sobre animación facial **[ESC99][GOT99][LEW99],** emplean el modelo propuesto por MPEG-4.

Algo similar ocurre con la descripción de la forma del cuerpo que se realiza a partir de BDPs (Body Definition Parameters) y de postura corporal definida a partir de BAPs (Body Animation Parameteres), los cuales hacen

referencia a los valores angulares de los grados de libertad de las distintas articulaciones. Ambos tipos de codificación están siendo empleados para implementar actores humanoides en diferentes aplicaciones y trabajos de investigación **[CAP00] [PRE02] [GUT02] [GUT03]**.

3.6.2.2 Animación de humanos en VRML.

VRML Humanoid **[HAN99]** es una propuesta para definir un estándar de representación de humanos mediante el empleo de VRML97 llevada a cabo por el Humanoid Animation Working Group. El objetivo de este grupo es definir una representación estándar para que humanoides creados por una determinada herramienta, puedan ser animados de distintas formas, por diferentes herramientas desarrolladas de forma independiente. Este estándar define un actor humanoide, fijando totalmente el número, nombre y topología de las articulaciones que lo forman (ver *Figura 3-4*). La primera versión de estándar (h-anim v1.0), proporcionaba una estructura esquelética del actor, que podía resultar excesivamente compleja para muchas aplicaciones. Esto ha sido corregido en su última revisión (h-anim 200x), en la que se proporcionan 4 definiciones del esqueleto alternativas, denominadas LOA (Level of Articulation), que permiten definir la estructura articulada empleando un menor número de puntos de articulación.

Un fichero VRML con especificación de humanoides contiene cinco tipos de nodos específicos que almacenan la información relativa el actor. Nodos *Joint*, que almacenan la información sobre una articulación, cada nodo *Joint* puede tener como hijos a otros nodos *Joint,* también nodos *Segment*, que describen la parte del cuerpo asociada con esa articulación (como por ejemplo un antebrazo o un muslo). Nodos *Site* que definen posiciones relativas a los segmentos, y que por ejemplo son utilizados como puntos auxiliares para colocar ropa o joyas. Cada nodo *Segment* tiene un conjunto de nodos *Displacer* que especifican conjuntos de vértices dentro del segmento, y que son empleados para controlar la deformación de los vértices de la superficie. También existe un nodo *Humanoid* que almacena características generales del actor tales como el autor o el copyright, y dispone de referencias a todos sus nodos *Joint* y *Segment.*

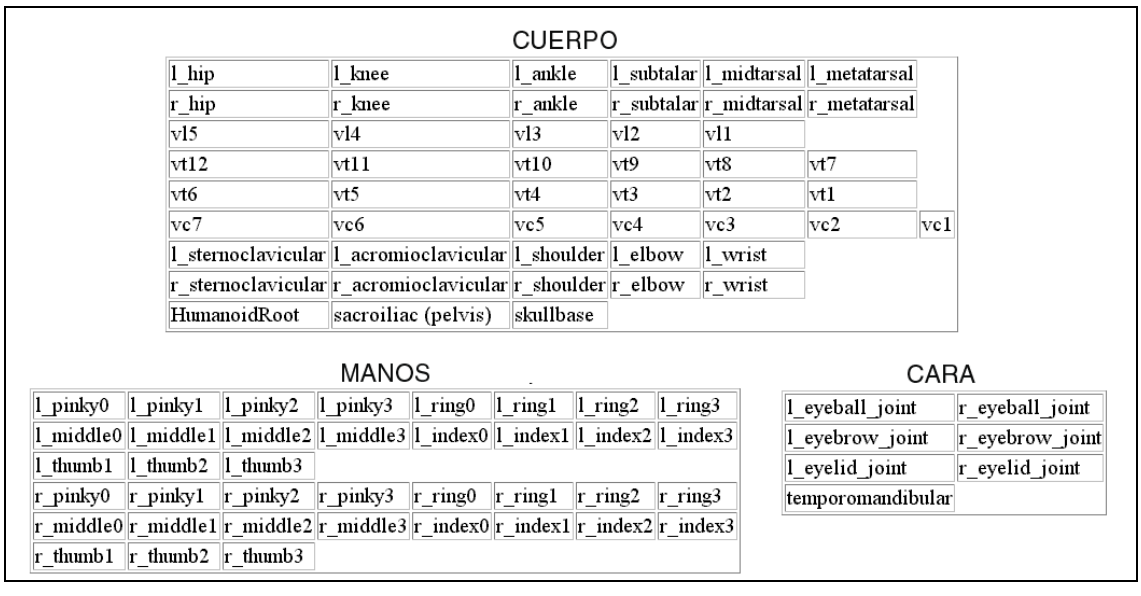


Figura 3-4.Convenio de nombres de articulaciones en Humanoid VRML97.

Estos cinco tipos de nodos han sido definidos a partir de nodos de tipo *PROTO* (ver apartado de VRML en el capítulo anterior). El estándar también contempla que el mismo fichero pueda contener secuencias animadas por Keyframing en las que las salidas de varios nodos VRML de tipo *Interpolator* son conectadas con las articulaciones del humano. También es posible que el fichero incluya nodos *Script* que accedan directamente a las articulaciones del humano mediante diferentes métodos de control.

3.6.3 Actores virtuales en simulaciones distribuidas.

Otra de las áreas en las que se han hecho importantes estudios ha sido en el empleo de actores en entornos virtuales distribuidos. En estos entornos, cada usuario es representado por medio de un actor virtual. La postura de un actor está definida a partir de los valores de los grados de libertad de sus articulaciones, y su transmisión por red necesita una cantidad considerable de ancho de banda, especialmente si en el entorno hay muchos participantes. La transmisión eficiente de esta información relativa a las posturas de los actores virtuales a través de la red es un problema que ha sido abordado desde diferentes ópticas.

El sistema *NPSNET* **[MAC94],** desarrollado en el *Naval Postgraduate School* en los Estados Unidos, integra actores virtuales humanos en un entorno *DIS* (*Distributed Interactive Simulations*) **[IEE93].** Su sistema incluye múltiples niveles de detalle, tracking del movimiento del cuerpo y un conjunto predeterminado de posturas **[PRA97]**. Los actores de *NPSNET* utilizan el modelo de humano *Jack* desarrollado en la *Universidad de Pensilvania* **[BAD93]** y emplean cuatro niveles de detalle que afectan al número de grados de libertad a tener en cuenta (50, 19, 12 y 0) en cada instante, y que es determinado en función de la distancia al observador. El modelo de animación a emplear también se adapta de a ese número de grados de libertad. La información referente a la parte inferior del cuerpo es transmitida recurriendo a un conjunto de posturas y secuencias predefinidas que son transformadas en el host que recibe la información en los correspondientes valores angulares. Los valores de las articulaciones de la parte superior del cuerpo son obtenidos mediante tracking directo de la cabeza y los brazos y son transmitidos por red utilizando campos del protocolo estándar de *DIS*, también llamados *PDUs* (*DIS protocol Data Units*). Este sistema también tiene la capacidad de representar grupos de humanos, haciendo que uno de ellos actúe como líder, y que el resto de actores reaccionaran en función de su comportamiento, esto hace que tan sólo baste con transmitir por red los datos del actor líder.

Una aproximación similar es la *VLNET* (*Virtual Life NETwork*) **[TOL97]**. La *VLNET* es también un sistema que permite que múltiples usuarios situados en distintos puntos puedan interactuar en tiempo real dentro de un escenario virtual común. Los participantes son representados como actores virtuales humanoides, y mediante ellos, interactúan con el entorno y el resto de participantes **[THA95a]**. El modelo de humano empleado utiliza 125 grados de libertad en el caso de utilizar un modelo sin manos y 175 en el caso de que éstas sean tenidas en cuenta. Agrupa los grados de libertad del cuerpo en 17 grupos, con 4 grupos adicionales para cada mano. Estos grupos están diseñados tendiendo en cuenta la importancia de los grados de libertad y sus interrelaciones. Cada uno de estos grupos puede ser enviado por separado, de tal modo que sólo se transfieren aquellos grupos que han sufrido alguna modificación en uno de sus DOFs. Para poder permitir esto, el bloque de datos relativo a la

postura de un actor va precedido por una mascara de 17 bits que indica que grupos se están transmitiendo. Este sistema permite también controlar el formato en el que se envían los datos de cada DOF, que pueden ser un float de 4 bytes, un entero de 2 bytes discretizado entre 0 y 360 o una representación de 1 byte, también codificada entre 0 y 360. Evidentemente la utilización de modelos de menos bytes introduce una perdida de precisión en la postura final del cuerpo del actor virtual. Emplea un codificador predictivo que procesa la información teniendo en cuenta los valores enviados previamente. Con este tipo de técnicas pueden conseguir unos ratios de compresión entre los márgenes 5:1 y 10:1 con unas perdidas de calidad que pueden ser asumibles. *VLNET* también permite que el cuerpo del actor sea transferido por la red en tiempo de simulación utilizando *asincronous progressive loading*, esto es empleado para traer nuevos participantes y usando para ello distintos modelos con diferente nivel de detalle. De este modo la interacción pueda comenzar con un cuerpo de baja calidad que va mejorando a medida que van llegando los modelos de más elevada calidad.

Para reducir la cantidad de transferencias de información en un entorno distribuido es posible utilizar un algoritmo conocido como *dead-reckoning* **[GOS94],** el cual en la actualidad es utilizado para gestionar el movimiento de objetos móviles no articulados en entornos como DIS **[IEE93]** y NPSNET [**MAC94]**. El método consiste en transferir no solamente la posición del móvil, sino parámetros que describan en más profundidad su movimiento, como por ejemplo su velocidad y aceleración. Una vez el ordenador remoto conoce estos valores puede calcular internamente la siguiente posición por extrapolación mediante la utilización de algún tipo algoritmo predictivo. El ordenador que envía los datos realiza simultáneamente esos cálculos sobre una copia "fantasma" del móvil original (*ghost model* en terminología inglesa). En el caso de que la diferencia entre las posiciones del móvil real y su fantasma supere un cierto umbral, se envía nuevos datos al ordenador remoto para que corrija su posición. Utilizando esta técnica, las transmisiones por red solamente se producen si es necesario realizar una corrección debido a que los valores fruto de la predicción se distancien de los reales. Así pues, es muy importante que el modelo empleado para realizar las predicciones se comporte de forma adecuada. En este sentido, el filtro de Kalman **[BRO92]**, parece presentar muy buenos resultados en el caso de que sea necesario realizar una predicción sobre una postura corporal **[AZU95] [FOX96]**. Algunos modelos, utilizan métricas para calcular el error entre posturas que permiten definir diferentes niveles de tolerancia en las diferentes articulaciones.

En actores virtuales, es posible aplicar *dead-reckoning* a nivel de los valores de los grados de las articulaciones del actor y también a nivel de parámetros de alto nivel que describen la acción que está realizando (por ejemplo la velocidad y dirección en el caso de la acción de caminar), en este último caso, resulta especialmente conveniente que existan mecanismos que sean capaces de sintetizar movimiento a partir de un conjunto reducido de parámetros de alto nivel **[BOU90] [UNU95] [TOS96]**.

3.7 Campos de aplicación.

A pesar de la reciente popularización de tecnologías relacionadas con la imagen 3D interactiva, y a pesar del nivel de dificultad que entraña definir y gestionar actores sintéticos en tiempo real, existen muchos campos en los que su utilidad es más que evidente y ésta es la causa por la cual se han desarrollado distintos tipos de actores virtuales orientados a cubrir necesidades específicas del mercado. Como norma general se puede decir que son desarrollos totalmente verticales que recurren a la utilización de grafos de escena e incluso a librerías de bajo nivel propias.

En este apartado se hace una relación de la mayoría de campos en los que se están aplicando actores virtuales, o bien de en áreas en las que dado el estado de las investigaciones se prevé que tengan una aplicación muy próxima.

3.7.1 Estudios de Ergonomía.

Uno de los primeros trabajos sobre actores virtuales interactivos fue el actor *Jack*, desarrollado originalmente por la *Universidad de Pensilvania* **[BADb02]**, y comercializado a continuación por diferentes empresas. *Jack* es un actor virtual humano que permite analizar la ergonomía de un determinado espacio. Desde su entorno de trabajo se puede importar gran cantidad de modelos provenientes del mundo del CAD, puede ser introducido en la maqueta virtual de un vehículo, y ser empleado para analizar la accesibilidad a sus controles, la visibilidad existente, y en general todas sus características ergonómicas (ver *Figura 3-5*).

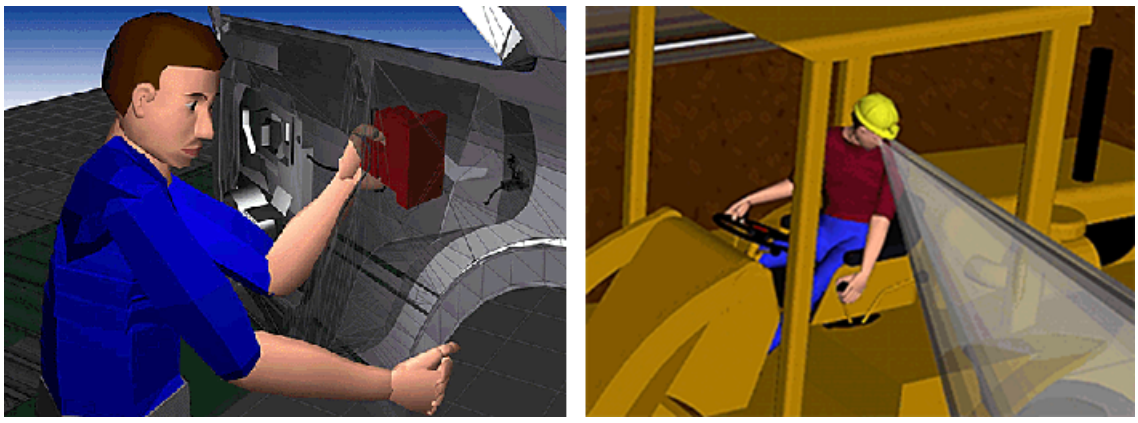


Figura 3-5. Estudios de Ergonomía con actor virtual Jack.

Este actor está formado por 88 articulaciones, 16 de las cuales se utilizan para el modelo de la mano y 17 para la columna vertebral, no presenta (ni necesita) una buena calidad gráfica y tampoco tiene requerimientos de tiempo real estricto (es suficiente con una respuesta interactiva de 8-10 frames/segundo). Es posible controlar la forma de su cuerpo mediante un conjunto variables antropométricas, y puede proporcionar información sobre su campo de visión, las fuerzas que puede aplicar, o el nivel de comodidad de una determinada acción. Tiene controles de

alto nivel que le permiten alcanzar y agarrar objetos, autoequilibrarse, detectar colisiones en tiempo real, y es extensible mediante programación en LISP.

Existen versiones de este software que permiten que haya también varias figuras humanas en mismo entorno, los que resulta especialmente útil para ensayar métodos cooperativos de reparación rápida de máquinas. Es un software totalmente vertical orientado hacia la ingeniería y que permite realizar un diseño orientado a la facilidad de acceso, la facilidad de reparación, seguridad, visibilidad, y reducción de riesgos.

3.7.2 Presentadores Virtuales para Televisión.

Los primeros trabajos sobre presentadores virtuales para Televisión no eran actores tiempo real sino en secuencias grabadas de vídeo realizadas con técnicas manuales más próximas a la animación 3D tradicional **[THA95b].** En la actualidad algunas televisiones utilizan actores sintéticos en tiempo real, este tipo de actores son los que en la actualidad requieren una mayor potencia de cálculo, puesto que por un lado se requiere una calidad gráfica muy elevada, lo que implica un elevado número de polígonos y la gestión en tiempo real de las deformaciones de piel, y por otro que es necesario mantener de forma constante un frame-rate elevado (50 imágenes/segundo en el caso de formato PAL).

Es habitual que para la gestión de un único actor virtual de este tipo sea necesario emplear una ordenador de varias CPUs y con un costoso sistema gráfico. En la *Figura 3-6* se muestra a *Cléo*, uno de los primeros actores sintéticos en tiempo real empleados en Televisión, así como un esquema descriptivo del sistema hardware que emplea, esta actriz fue desarrollada por el MEDIALAB de París para Canal Plus Francia.

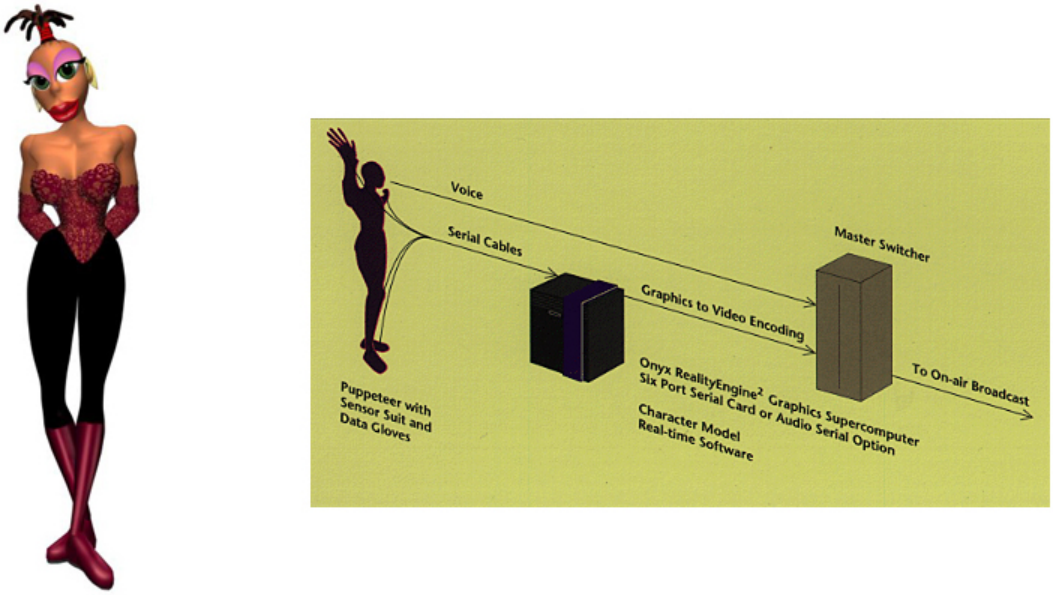


Figura 3-6. Presentadora virtual Cléo y sistema hardware empleado.

El control de estos actores virtuales se realiza recurriendo a sistemas de captura de movimiento que analizan la posición de un actor real y la mapean en tiempo real sobre el actor virtual, un sistema distinto se encarga de gestionar la animación facial mediante el empleo de guantes de datos o sistemas de reconocimiento de la expresión facial.

3.7.3 Presentadores Virtuales Autónomos.

En la actualidad existe, también, una búsqueda de un tipo de actor virtual con un comportamiento autónomo, que actúe como un nuevo tipo de interfaz persona-computador. Estos actores virtuales, pueden desarrollar el papel de ayuda en un programa de ordenador, asistir a la compra de un producto por Internet, o en un puesto de información multimedia.

En la mayoría de casos el actor se ve reducido a tan sólo la cabeza, por ello, en literatura inglesa a este tipo de aplicaciones les llaman "Talking heads". Es habitual que estos actores utilicen técnicas que les permiten transformar texto escrito en voz, y controlar de forma sincronizada los movimientos de los labios **[NAG94][SPR95]**. En alguno de estos programas se intenta que exista la posibilidad de reconocimiento de voz.

Existen empresas cuya principal línea de trabajo es el desarrollo de este tipo de aplicaciones. Resulta ilustrativo el caso de Redted, una empresa que vende varios productos relacionados con Talking Heads, como por ejemplo un control ActiveX para que los desarrolladores de Windows puedan incorporarlas en sus aplicaciones o un applet de Java que permite incorporar cabezas parlantes en una página web.

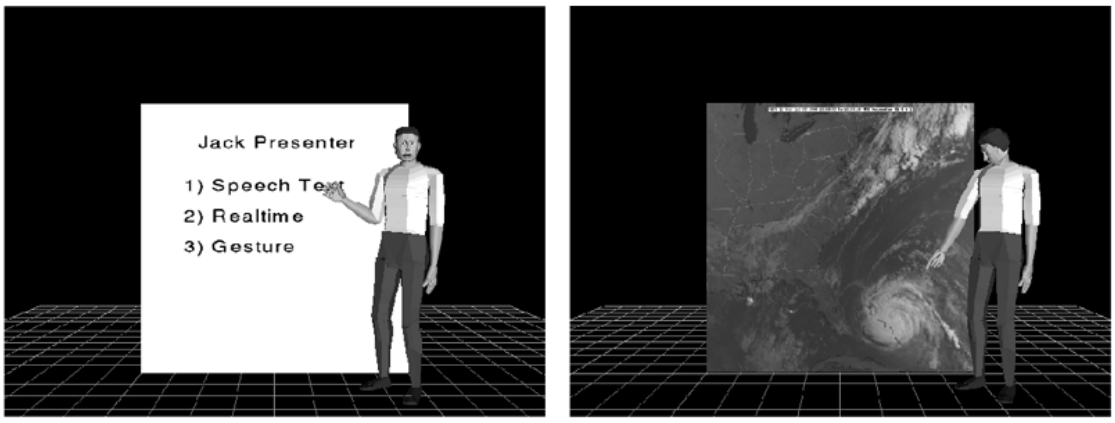


Figura 3-7. Imágenes del presentador virtual autónomo Jack.

Existen tambien trabajos como *Jack Pressenter* **[NOM97],** que tratan de gestionar un presentador virtual completo (no sólo la cabeza) moviéndose delante de una pantalla o pizarra, prestando especial atención a la generación de movimiento sincronizado con la expresión oral, a los mecanismos que controlan los gestos manuales que dirigen la atención del publico hacia una parte concreta de la pizarra, a los que hacen adoptar la postura corporal más adecuada para la presentación de una determinada información existente en la pantalla, o los sucesivos cambios de atención entre el publico y el contenido de la pantalla que tiene detrás (ver *Figura 3-7*). Los datos de entrada al presentador virtual consisten en el texto que el presentador ha de pronunciar, el cual lleva

además incorporados ciertos comandos de control, la mayoría de ellos referentes al lenguaje corporal del presentador.

3.7.4 Streaming de actores.

Si disponemos de un actor virtual que actúe hombre del tiempo personalizado, la gente podrá conocer la predicción meteorológica tan pronto como estén los datos disponibles **[NEG95].** Esta frase de Negroponte describe uno de los objetivos finales objetivo del streaming de actores, disponer en nuestra casa de presentadores virtuales personalizados que pueden presentarnos la información que están recibiendo en tiempo real a través de Internet. En la actualidad existen distintas iniciativas empresariales que buscan una línea de negocio en este campo (Ejemplos en la *Figura 3-8*).

Figura 3-8. Ejemplos de aplicaciones con streaming de actores virtuales.

Este tipo de actores presentan características similares a los presentadores del apartado anterior, pero a diferencia de estos, están totalmente orientados hacia Internet o televisión interactiva, no siempre son autónomos (se puede estar enviando información que está siendo generada interactivamente por un actor real en el centro emisor), y no siempre tienen la capacidad de generar voz a partir de texto escrito, de hecho lo habitual es hacer simultáneamente streaming tradicional de audio y de la expresión facial. Este tipo de actores también presenta muchas similitudes con los actores empleados para videoconferencia 3D que serán tratados en el siguiente apartado, siendo la principal diferencia el hecho de que en estos últimos la comunicación es bidireccional y los procesos de codificación, transmisión y reproducción han de estar produciéndose de forma simultanea.

Futuras modificaciones de este tipo de aplicación, harán posible la retransmisión de pruebas deportivas por Internet, con lo cual el espectador tendrá además la capacidad de visualizar a los jugadores desde el punto de vista que desee.

3.7.5 Sistemas de Video-conferencia 3D.

Desde hace varios años han existido varias líneas de trabajo centradas en la búsqueda de sistemas de videoconferencia de características adecuadas para ser empleado de forma genérica. Casi todos los esfuerzos han estado centradas en lograr buenos sistemas de compresión de vídeo, sin embargo, desde hace algunos años se

está intentando conseguir sistemas de videoconferencia que empleen tecnología relacionada con la informática gráfica tiempo real. La idea principal es relativamente sencilla, consiste en transmitir en los instantes iniciales de la comunicación una fotografía y un modelo 3D de la persona. Un sistema de visión artificial o incluso un sistema mecánico analiza la expresión de la cara de los interlocutores, y la codifica en función de un conjunto muy pequeño de parámetros (elevación y orientación de los párpados, apertura de la boca, etc.) **[PET88] [GOL94]**. Estos parámetros son trasmitidos por la red, y mapeados sobre el modelo 3D que representa a cada uno de los interlocutores (en la *Figura 3-9* se puede observar un sistema desarrollado por el MIRAlab que realiza en tiempo real las actividades de procesamiento de la imagen, extracción de parámetros, y mapeado sobre el modelo virtual **[MAG98]**).

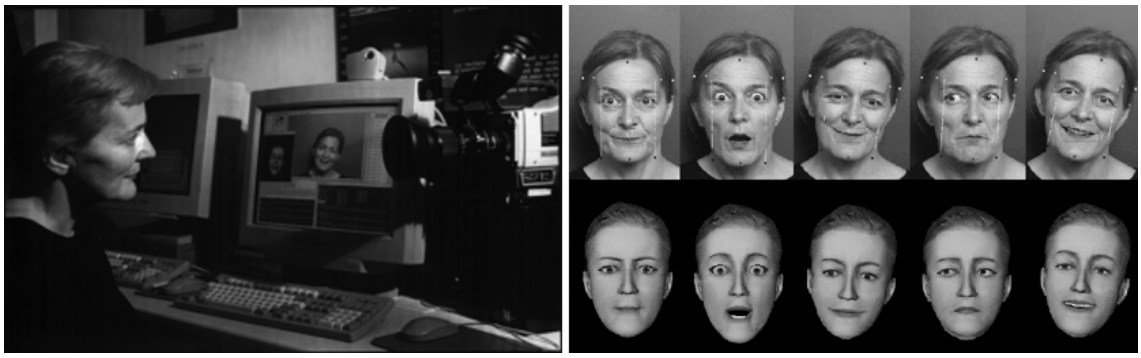


Figura 3-9. Sistema de procesamiento de tiempo real de la expresión facial del MIRAlab.

Algunos sistemas, proponen la utilización de una geometría básica predefinida (formada por ejemplo a partir de anillos rígidos de vértices **[WIN97]**) sobre la que se adapta la geometría correspondiente al usuario que ha sido obtenida a partir de algún sistema de captura de información 3D **[FAL94]**, Sobre esta geometría capturada, se aplica como textura una imagen del interlocutor que también puede haber sido obtenida automáticamente a partir del propio sistema de captura de información 3D, o bien por otros medios **[NIE95].** En este tipo de sistema la geometría básica está vinculada con un esqueleto interno, la modificación de este esqueleto se propaga a la geometría escaneada que representa al interlocutor, adoptando la posición correspondiente. La posición del interlocutor en cada instante puede ser determinada de varios modos, siendo los más adecuados para este tipo de aplicaciones aquellos basados en el reconocimiento de imagen **[KRO94][TES95][GRA96].**

3.7.6 Avatares y Agentes en entornos virtuales distribuidos.

Una interesante aplicación de los actores virtuales es su utilización en entornos virtuales distribuidos a través de Internet. Un entorno virtual distribuido (en literatura inglesa *NVE: Networked Virtual Enviroment*) puede ser definido como un entorno virtual único controlado por un solo host, el cual es compartido por los diferentes participantes conectados desde diversos hosts. El software local de cada participante, recibe por red y almacena el conjunto total, o la parte del escenario que está utilizando, y emplea un actor virtual que lo representa para moverse por la escena, renderizandose las imágenes que se corresponden a su visión subjetiva.

Un avatar es una representación simbólica en un mundo virtual de una persona real. Por medio de un avatar, el usuario del sistema puede moverse por el mundo virtual, viendo lo que avatar ve, y haciendo que el avatar actúe según sus órdenes. La misión de los avatares **[CAP98]** en estos entornos es múltiple:

- Percepción. Se conoce si hay alguien alrededor.
- Localización. Se identifica la posición en la que está la otra persona.
- Identificación. Se puede reconocer a esa persona.
- Visualización de otras características de interés (por ejemplo se ve a que actividad se está dedicando la otra persona, o cual es el que grupo de gente al que está hablando).

En estos entornos virtuales distribuidos, también se utilizan otro tipo de actores virtuales que no son controlados por ninguna persona real, y a los que se les denomina *agentes*. Los agentes están dotados de un control totalmente autónomo y se suelen utilizar para proporcionar ayuda al usuario.

Figura 3-10. Vistas de aplicaciones de 3D-Chat (ActiveWorlds, Blaxxum y Worlds).

En los sistemas comerciales existentes en la actualidad (ver *Figura 3-10*), el medio principal de comunicación entre los avatares son mensajes de texto, y la calidad gráfica y nivel de interacción es aún muy baja, esto hace que sean en realidad se comporten como un tipo especial aplicaciones de chat multiusario (de hecho, es habitual recurrir al término 3D-Chat para referirse a este tipo de aplicación). En estos sistemas, los avatares son actores semi-autónomos a los que el usuario le indica la dirección en la que ha de moverse, existiendo comandos de alto nivel que le permiten acceder a una librería de posturas corporales, y expresiones faciales pregrabadas.

3.7.7 Entornos Colaborativos inmersivos.

Son sistemas en los que varias personas emplean periféricos de realidad virtual para entrar en un entorno virtual compartido, con el objeto de realizar una tarea común. En estos entornos la calidad de la imagen, los movimientos y la interacción son prioritarios. Se suele realizar tracking directo de la postura facial y corporal, y la comunicación es a través de la voz.

Existe la posibilidad de que cada usuario este entrando desde un punto distinto de la red, en este sentido tienen características comunes con los sistemas de videoconferencia 3D y los sistemas de 3D-Chat. La diferencia principal con los sistemas de 3D-chat es la inmersión, y la calidad gráfica y de interacción. A diferencia de los

sistemas de videoconferencia 3D, en los que toda la comunicación se realiza por medio de una pantalla de ordenador, en estos sistemas los interlocutores utilizan distintos tipos de periféricos de realidad virtual que les producen la sensación de estar inmersos en el entorno 3D común, que comparten con las otros interlocutores. En ese espacio de encuentro podría haber objetos auxiliares sobre los realizar algún análisis común, como por ejemplo el prototipo virtual de un nuevo automóvil o la maqueta de un edificio.

Figura 3-11. Partida de tenis virtual en un entorno colaborativo.

En la actualidad los trabajos en este campo están siendo realizados a nivel experimental en universidades y centros de investigación. Como ejemplo ilustrativo de este tipo de aplicaciones se puede citar el interesante trabajo realizado por el *MIRAlab* de la *Universidad de Genova* y el laboratorio de gráficos por computadora de *Laussane* **[MOL99]**. En este experimento (ver *Figura 3-11*), dos personas vestidas con cascos de realidad virtual, sensores magnéticos de posición y guantes de datos jugaron una partida virtual de tenis estando cada una de ellas en una ciudad distinta.

3.7.8 Interacción en primera persona.

En las aplicaciones de realidad virtual en las que se utiliza casco de visualización, el usuario pierde la visión de su propio cuerpo. En estos casos se hace necesaria la utilización de un modelo 3D que representa el usuario dentro del entorno virtual, y de sistemas de captura de movimiento corporal y guantes de datos que obtengan en tiempo real la posición de sus articulaciones.

El uso de un actor virtual que represente en todo momento la posición del usuario se hace totalmente necesaria en aquellos circunstancias en las que el usuario entra en contacto con los objetos virtuales. En estos casos, es necesario realizar detección de colisiones entre dichos objetos y el actor virtual, y, además, es necesario disponer de una realimentación visual adecuada (imaginemos por ejemplo que fuese necesario manejar un teclado virtual, y no se tuviese una representación 3D de las manos).

3.7.9 Monitorización.

La forma de moverse de animales y personas puede ser almacenada como información tridimensional para ser posteriormente analizada de forma detallada. Estos movimientos son almacenados en estructuras que luego se

aplican sobre actores virtuales, el usuario puede seleccionar el ángulo de visión más adecuado, y reproducir la secuencia a estudiar tantas veces como sea necesario. Este tipo de sistemas resulta de especial utilidad en el análisis de movimientos realizados por deportistas, o en el estudio del comportamiento de masas.

3.7.10 Videojuegos.

La industria de los videojuegos fue una de las primeras en aprovechar las características de los actores virtuales, y quizá por ello es una de las que presenta actores virtuales más avanzados, especialmente en los aspectos referentes a gestión de comportamiento, y a la rapidez de dibujado. Suelen emplear altas frecuencias de refresco (es común que alcancen tasas de 50 frames/segundo), y trabajan con grafos de escena propios (u otros sistemas de representación no basados en un grafo de escena), algunos utilizan librerías de bajo nivel standard como OpenGl o Direct3D, y otros están orientados a plataformas hardware especiales (vídeo-consolas). Su principal inconveniente es que son desarrollos verticales muy optimizados, pero poco reutilizables.

La elevada importancia económica del sector de los videojuegos en los últimos años, ha afectado de diversos modos a la informática gráfica en tiempo real, haciendo muy rápido el ritmo de mejoras experimentado por el hardware gráfico, incrementando considerablemente la documentación referente a la implementación de este tipo de aplicaciones, y mejorando la capacidad y flexibilidad de las herramientas empleadas en la creación de los videojuegos (3d-engines o game-engines). Esta influencia ha alcanzado también a la comunidad científica, proporcionando nuevas herramientas de investigación **[LEW02],** y siendo tenida en cuenta incluso en los planes educativos **[GAL00]**. En la actualidad existen diversos casos en los que en los que las investigaciones sobre el comportamiento de los actores virtuales han sido desarrollados empleando como base los motores de juegos comerciales como el Unreal Tournament **[LOZ04]** o el Quake**[LAI01]** .

Respecto a su representación geométrica, la utilización del modelo de geometría rígidas interconectadas era común en los primeros videojuegos 3D, en la actualidad, es habitual la utilización de *skining*.

Existen juegos que permiten que varios jugadores participen simultáneamente en un mismo entorno 3D, bien sea a través de redes locales o de Internet, actuando de forma similar a la descrita en los entornos distribuidos. Algunos videojuegos tienen capacidad para gestionar simultáneamente un elevado número de actores con comportamientos que presentan complejas interrelaciones (por ejemplo en un juego de fútbol).

En general se puede decir que el elevado nivel de acabado de los actores 3D para videojuegos es fruto de la verticalidad de sus desarrollos.

Figura 3-12. Actores virtuales en Videojuegos.

En la *Figura 3-12*, se pueden observar algunos de los videojuegos que recurren a la utilización de actores virtuales más populares (por orden Virtua Fighter, Mario Bros, Quake y TomRaider).

3.7.11 Simulación Militar. Di-Guy

El producto comercial de referencia en actores virtuales para simulación militar es *Di-Guy*, desarrollado por *Boston Dynamics* **[BDI96],** una empresa surgida como spin-off del laboratorio de Inteligencia Artificial del *MIT*. *Di-Guy* es un software que proporciona un conjunto de soldados virtuales diseñados para actuar en entornos de simulación en tiempo real. Dispone de un sistema de control de movimiento, que gestiona de forma interna las tareas más comunes de un soldado tales como pueden ser caminar, correr, arrastrarse, esconderse, disparar o morirse, encargándose también de gestionar automáticamente las transiciones entre dichas actividades.

Los actores proporcionados por *DI-Guy* tienen una estructura topológica cerrada (son siempre humanos de 16 puntos de articulación y 54 grados de libertad), emplean un modelo de representación geometrías rígidas interconectadas, e incorpora gestión de niveles de detalles (ver *Figura 3-13*). Pueden ser integrados en aplicaciones que estén basadas en el empleo de *OpenGl Performer* y otros grafos de escena.

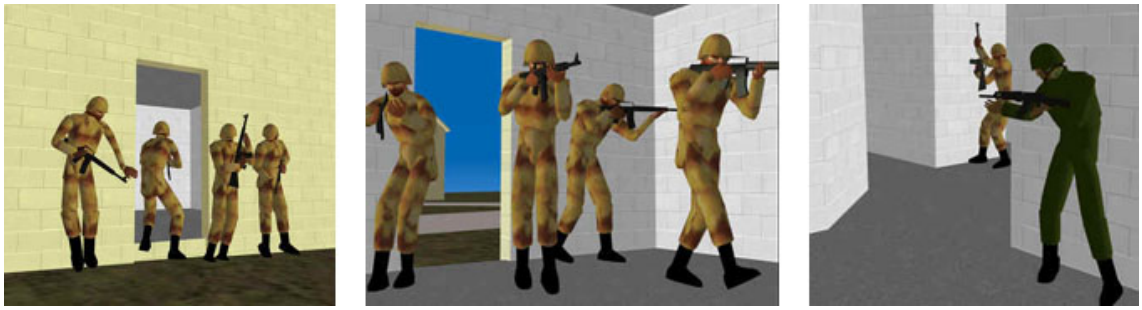


Figura 3-13. Soldados virtuales empleados en el programa de simulación militar DI-Guy.

En este tipo de aplicación, el programador accede a un API de alto nivel que gobierna el sistema de control interno que gestiona automáticamente los movimientos de los actores, el usuario no tiene que preocuparse de definir dichos movimientos, como contrapartida, el conjunto de movimientos se reduce al proporcionado por la herramienta.

Este software ha sido desarrollado con una orientación clara hacia la simulación y entrenamiento militar, esto hace que presente muy buenas características en cuanto a frame-rate y a integración en entornos de simulación distribuida como *DIS* **[IEE93],** o *HLA* **[DMS96]**. En este tipo de aplicaciones resulta de especial interés la capacidad de reconocer ordenes de alto nivel del estilo "corre" o "detente", de comandar un grupo de soldados a través de ordenes a su líder, o de emplear de las técnicas de *dead-reckoning* como las descritas en el apartado 3.6.3. Todas estas capacidades hacen que la necesidad de ancho de banda se reduzca considerablemente.

En los últimos años este software se ha adaptado a las necesidades del mercado de simulación civil y se han desarrollado modelos de deportistas y peatones.

3.7.12 Simulación Civil.

Si bien en un principio las tecnologías de simulación tridimensional se aplicaron en el campo militar, su utilización es ahora muy común en otros campos totalmente distintos. En este sentido se han desarrollado simuladores de diversos tipos de vehículos y maquinaria civil, y se han utilizado estas mismas tecnologías para realizar paseos interactivos por entornos arquitectónicos, o por reconstrucciones históricas. Aún hoy, la mayoría de este tipo de aplicaciones no emplean actores virtuales, sin embargo es totalmente deseable que en un simulador de conducción existan peatones o que haya personas y animales que lo habiten, lo mismo ocurre conlos entornos arquitectónicos y reconstrucciones históricas. La incorporación de actores virtuales a aplicaciones de simulación civil, por un lado enriquece su contenido, y por otro lado expande enormemente su campo de aplicación, permitiendo que se desarrollen aplicaciones en las que los actores (y no las máquinas o los entornos) sean el elemento principal de la simulación, tal sería el caso de algunos softwares de simulación para tratamientos psicológicos, de aquellos orientados a la planificación de movimientos coreográficos, o de estrategias deportivas, a la planificación de espectáculos teatrales o de escenas cinematográficas.

3.8 Conclusiones.

Se ha presentado el estado del arte de los actores virtuales, caracterizado por la utilización de arquitecturas multicapa (comportamental, motora, geométrica), y la existencia de varios campos de aplicación en los que ha existido una tendencia a realizar desarrollos de tipo vertical. La conexión entre las estructuras y métodos empleados para implementar actores virtuales, y la forma de operar de los grafos de escena actuales ha sido mínima. La gran mayoría de las investigaciones han estado centradas en el modelado comportamental. El interés por los modelos motor y geométrico ha sido muy reducido.

Las conexiones existentes entre el Modelo Comportamental de los actores virtuales y otras áreas científicas tales como la Inteligencia Artificial, Psicología, Etología, Neurologia etc. ha despertado el interés de muchos investigadores, con la consiguiente aparición de multitud de publicaciones en esta área. Es norma general que estas investigaciones presten poca atención a los modelos motor y geométrico y a su integración en sistemas de simulación en tiempo real.

El Modelo Motor ha recurrido en la gran mayoría de los casos a la utilización de movimientos grabados, siendo muy habitual la utilización de técnicas de *motion capture*, en ese sentido se han realizado grandes avances en los sistemas de captura de la posición corporal. Sin duda, la mejor forma de implementar el modelo motor de un actor sería la síntesis de movimiento, sin embargo, la dificultad actual para conseguir movimientos realistas, estables y compatibles con el tiempo real hace que las técnicas sean en la práctica muy poco habituales.

El Modelo Geométrico es la capa de más bajo nivel dentro de la implementación de los actores virtuales, y, quizá por ello, a la que se ha prestado menos atención. Sin embargo es la capa que forzosamente ha de estar en cualquier actor virtual (ya sea autónomo, semi-autómomo o de control directo), y es el aspecto de mayor importancia a la hora de integrar los actores en una aplicación de simulación. En el estado del arte se han presentado las formas de descripción de la geometría más habituales, basadas en geometrías rígidas interconectadas, y en clústering de vértices, así como otras técnicas más complejas con las que consiguen resultados más realistas. En general se puede decir que estado del arte del modelo geométrico ha estado caracterizado por la falta de preocupación por el rendimiento y la estandarización. Los únicos esfuerzos en este sentido han sido realizados en relación con el desarrollo de estándares para la parametrización de la postura de actores de tipo humanoide, y también en el estudio de la adecuada codificación de la postura de los actores, para su utilización en simulaciones distribuidas.

Ha existido una tendencia a crear los actores virtuales y sus mundos siguiendo una filosofía Top-Down, según la cual, se ha modelado primero su inteligencia, luego un cuerpo que pudiese mostrar las actividades ideadas por esa inteligencia, y por último un entorno 3D en el que ese cuerpo pudiese realizar sus actividades. Esa orientación es la causa de que la gran mayoría de actores virtuales actuales vivan encerrados en sus propias aplicaciones. El punto de vista de un desarrollador de aplicaciones de simulación es totalmente el contrario, él ya dispone de un substrato estable para desarrollar sus escenarios 3D (mediante grafos de escena), y desea, en

primer lugar, disponer de actores virtuales que presenten una representación geométrica compatible con esos entornos, y, en segundo lugar, dotar a esos actores de la mayor capacidad de inteligencia posible.

En este trabajo se presentará esta segunda orientación, partiendo del estado del arte de los gráficos en tiempo real descritos en el capítulo anterior, para definir nuevas estructuras y métodos que, agregados sobre una estructura de *Grafo de Escena*, permitan definir el modelo geométrico de los actores virtuales de un modo estandarizado y computacionalmente eficiente. Las estructuras y métodos propuestos estarán diseñados de modo que proporcionen un buen substrato, para el desarrollo de los modelos Motor y Comportamental.

PARTE III
ESTRUCTURAS Y MÉTODOS

Capítulo 4. Integración de Actores Virtuales en el Grafo de Escena TR.

4.1 Índice.

4.1 Introducción.

Para la representación de una escena tridimensional en tiempo real, es necesario emplear un hardware gráfico especializado que sea gestionado mediante una librería gráfica de bajo nivel. Es posible diseñar una aplicación con gráficos 3D en tiempo real a partir de la utilización directa de una estas librerías, sin embargo, las librerías gráficas de bajo nivel están tan centradas en la adecuada gestión del hardware gráfico, que si se necesita desarrollar una aplicación de una complejidad media resulta imprescindible emplear alguna librería con un mayor nivel de abstracción.

En la actualidad, existen varias librerías gráficas de alto nivel orientadas a la definición y dibujado en tiempo real de escenas tridimensionales complejas. A pesar de las diferencias existentes entre dichas librerías, todas ellas tienen en común el hecho de estar organizadas en torno a una estructura de nodos ordenados en forma de grafo dirigido acíclico a la que se denomina ***Grafo de Escena***. Los *grafos de escena*, además de simplificar el acceso a las librerías de bajo nivel, contribuyen a establecer una organización lógica de la escena, facilitan el control por parte del desarrollador, actúan como soporte para métodos de culling y gestión de niveles de detalle, y se emplean para establecer dependencias jerárquicas entre diferentes sistemas de coordenadas.

Un actor virtual puede ser definido directamente mediante la utilización directa de una librería gráfica de bajo nivel. Pero dada la complejidad que encierra la gestión de los actores virtuales, y el estado del arte de los gráficos en tiempo real, resulta mucho más apropiado buscar una solución que se base en la utilización de la estructura del *grafo de escena*.

Se podría definir la estructura articulada de un actor virtual recurriendo a utilización de los *nodos de transformación afín* existentes en los *grafos de escena* actuales, sin embargo, estos nodos han sido concebidos para tareas más sencillos, tales como ubicar un objeto en un determinado lugar, o bien para controlar el movimiento de un objeto simple como puede ser un coche o un avión. En general, se puede decir que un actor virtual puede ser ensamblado a partir de este tipo de nodos, pero su construcción resulta complicada, y el resultado obtenido es de difícil comprensión, su gestión resulta complicada y computacionalmente sería poco eficiente. Por otro lado, es necesario proporcionar una cierta encapsulación, que permita hacer que un nodo *Actor* sea algo distinto a un conjunto de nodos dispersos por el *grafo de escena*.

Los *grafos de escena* actuales están principalmente orientados a la representación de bases de datos estáticas (terrenos, edificios, etc.), o con objetos que presenten movimientos simples (básicamente vehículos). Tanto los tipos de nodos empleados, como los métodos de culling y gestión de LOD son adecuados para este tipo de escenarios, pero resultan insuficientes a la hora de definir y gestionar una escena en la que aparezcan varios actores virtuales, de hecho, puede ocurrir que una aplicación con un único actor virtual supere en

complejidad a una aplicación que implemente un simulador de vuelo. Imaginemos la complejidad de una simulación que contenga un centenar de actores virtuales.

Este trabajo analiza la problemática de la integración de actores virtuales en un grafo de escena, y propone, entre otras cosas, la utilización de dos nuevos tipos de nodos específicamente diseñados para la definición y gestión de actores virtuales. Estos dos nuevos tipos de nodos, denominados ***Actor*** y ***Skeleton*** están relacionados entre sí, y conjugan la capacidad de dotar al usuario de un control de alto nivel sobre los actores virtuales con una gestión de bajo nivel que reduzca su elevado coste computacional. El ***nodo Skeleton*** almacena la información referente a un punto de articulación de un actor, y actúa como elemento único a partir del cual es posible definir toda su estructura topológica. El ***nodo Actor*** define la posición del actor dentro de la escena, actúa como nodo padre para el resto de nodos que componen el actor, y actúa como centro de control del fragmento de *grafo de escena* que define al actor virtual.

En los siguientes apartados se expone en detalle todo el análisis realizado para la definición de estos dos nuevos tipos de nodos. Mostrándose todas las características de los nodos *Skeleton* y *Actor,* y demostrando su capacidad para actuar como elementos modulares básicos a partir de los cuales es posible ensamblar cualquier tipo de actor virtual de una sencilla, rápida, reutilizable y computacionalmente muy eficiente.

4.2 Nodos específicos para la gestión de actores. Análisis.

La definición de un actor virtual partir de los nodos tradicionales de un *grafo de escena,* genera como resultado estructuras complejas difíciles de manejar y computacionalmente poco eficientes. En este apartado se hace un primer análisis sobre cuales serían los nuevos tipos de nodos sería necesarios para subsanar estos problemas.

La solución al problema es analizada observando a los actores virtuales desde una doble óptica: como una estructura articulada de tipo genérico a la que hay que tratar de un modo eficiente, y como un ente encapsulado al que hay que poder tratar con un cierto nivel de abstracción. Desde el primer punto de vista se busca un método que permita la definición de la estructura articulada del actor y se propone una solución basada en la utilización de un único tipo de nodo, al que se denomina *Nodo Skeleton*. Desde la segunda óptica, se busca un método que permita tanto ubicar al actor en la escena, como realizar un control indirecto de todos los nodos que lo forman, y se propone una solución basada en la utilización de un nodo de nombre *Nodo Actor* que actúa como padre de todos los nodos que componen al actor virtual.

En este apartado se describen cuales han sido los criterios empleados para la definición de ambos tipos de nodos, y también se analiza la forma en la los dos nuevos tipos de nodos pueden ser integrados sobre un *grafo de escena* tradicional.

4.2.1 Representación de una estructura articulada sobre una Grafo de Escena. Bases para la definición del Nodo Skeleton.

En los paquetes de animación 3D de actores virtuales orientados a la industria cinematográfica es común emplear representaciones de la estructura articulada de un actor virtual basadas en la utilización de dos tipos de elementos: al primero de ellos se le suele denominar *Junta*, y hace referencia al un punto de articulación, y al segundo, que hace referencia a la distancia entre dos puntos de articulación se le suele denominar *Segmento* (ver *Figura 4-1*). La solución más inmediata al problema de representación de un actor virtual en una aplicación de simulación, sería directamente, trasladar este tipo de representación al *grafo de escena,* definiendo para ello unos nodos de tipo *Junta* y *Segmento*. En la *Figura 4-2* muestra el aspecto que tendría el fragmento de *grafo de escena* que representaría la pierna de un actor humano, a partir de estos hipotéticos nodos. Los nodos con información poligonal, que no aparecen representados en la figura, podrían estar indistintamente vinculados al nodo *Segmento* o al nodo *Junta*.

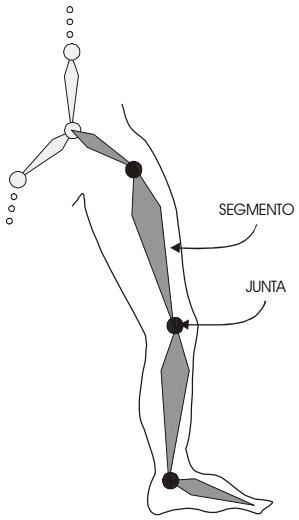


Figura 4-1. Representación tradicional de una estructura articulada de un actor formada por parejas *Segmento-Junta*.

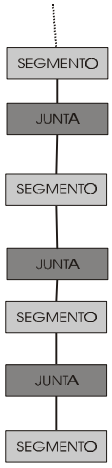


Figura 4-2. Representación jerárquica de un fragmento de un actor formada por hipotéticos nodos *Segmento* y *Junta*.

La definición de la estructura articulada de un actor por medio de pares *Segmento-Junta* resulta gráficamente muy intuitiva, especialmente porque existe una correspondencia visual entre los *Segmentos* de la estructura articulada de un actor virtual, y los huesos de un animal real. En el software de animación 3D tradicional, es habitual que el usuario defina y maneje a los actores actuando directamente sobre los *Segmentos*, quedando la existencia de las *Juntas* de alguna forma enmascarada. Desde el punto de vista de un *grafo de escena* para tiempo real, la óptica es totalmente la contraria, la información realmente importante es la que define las matrices de transformación que se han de aplicar, y que vienen definidas básicamente por los grados de libertad (dofs) contenidos en la *Junta*. La utilidad de las *Juntas* dentro del *grafo de escena* es obvia, sin embargo, la misión de los *Segmentos* no es tan evidente. Haciendo un pequeño análisis de las características de los *Segmentos* se puede observar lo siguiente:

- Un *Segmento* depende directamente de una sola *Junta*.
- Cualquier *Junta* tiene al menos un *Segmento* que depende de ella (si bien puede tener más).

- Hay *Segmentos* terminales que tienen uno de sus extremos sin ninguna *Junta*.
- Se emplean para definir la separación existente entre *Juntas* (es decir, actúa de offset entre los sistemas de referencia de las *Juntas)*.
- Actúan como elemento gráfico para la modificación de los valores de las *Juntas*. Usualmente se modifican los valores de los grados de libertad de la *Junta* seleccionando uno de los *Segmentos* que dependen directamente de ella.

Como resultado de estas observaciones se puede concluir que un S*egmento* presenta dos funciones: Actuar como ***offset*** sobre una *Junta* (indica la posición relativa en la que está respecto a la *Junta* de la que depende jerárquicamente), y actuar como ***elemento de interface*** para interactuar sobre los *dofs* de *Junta* de la que depende. Si bien es usual que los *Segmentos* desempeñen esta doble misión, existen algunos *Segmentos* que tan sólo actúan como *offset,* y otros que tan sólo actúan como *elementos de interface*. En la *Figura 4-3* se puede observar un ejemplo típico en el que existen *Segmentos* de estos tres tipos: La cadera es un caso en el que el *Segmento* es utilizado básicamente para indicar la separación entre el punto de unión de la columna vertebral con el coxis, y el punto de articulación de la cadera. Un caso muy claro en el que los *Segmentos* actúan tan sólo como *elemento de interface* es el de aquellos existentes en las terminaciones de las extremidades, tal es el caso del *Segmento* vinculado al pie.

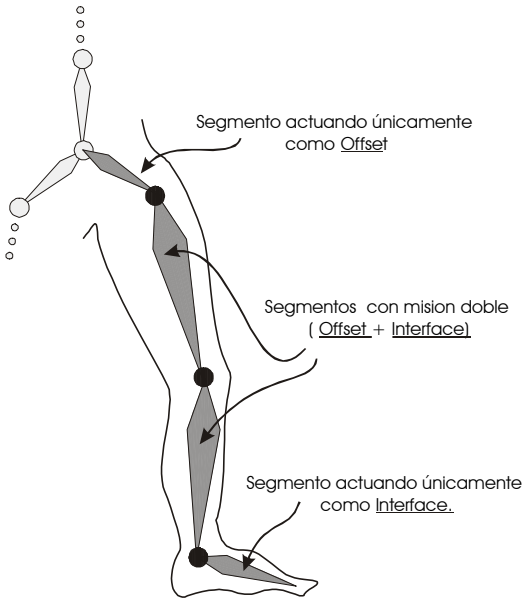


Figura 4-3. Hipotéticos nodos Segmento y su doble misión como Offset e Interface.

En definitiva, se puede considerar que los *Segmentos* actúan como elementos que proporcionan información auxiliar a las *Juntas.* Teniendo en cuenta esto, y con el objetivo de reducir al máximo el número de nodos necesarios para representar la estructura articulada de actor, se ha decidido definir una única estructura denominada ***nodo Skeleton,*** la cual almacena toda información referente a la jerarquía esquelética de un actor, y contiene la información tradicionalmente almacenada mediante estructuras de tipo *Segmento* y *Junta*. Los nodos *Skeleton* contienen la información sobre un punto de articulación y sus grados de libertad, también el ***offset*** existente con respecto a su *Skeleton* padre, así como un elemento gráfico que actúa de ***interface***.

En la Figura 4-4 se pueden observar con claridad los diferentes elementos constitutivos de un *nodo Skeleton.* También se puede observar como la estructura jerárquica de una pierna de un actor a partir de 3 hipotéticos *nodos Junta* y 3 hipotéticos *nodos Segmento* (los cuales además no se comportaban de una forma uniforme), mostrado en la Figura 4-1, ha sido sustituida por una estructura formada por tres únicos *nodos Skeleton* con un comportamiento estandarizado.

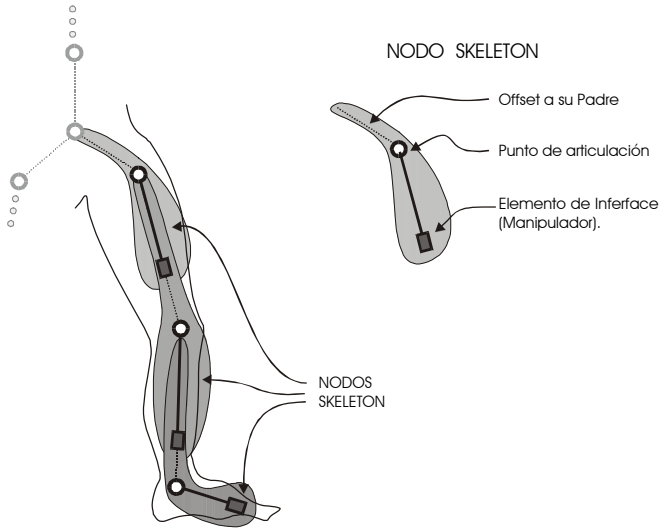


Figura 4-4. Representación de un fragmento de la estructura articulada de un actor mediante nodos *Skeleton*. Elementos constitutivos de un nodo *Skeleton*.

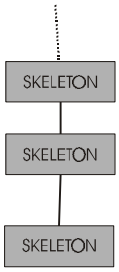


Figura 4-5. Representación jerárquica de un fragmento de un actor compuesto tan sólo de nodos *Skeleton*.

En la *Figura 4-4* y en la *Figura 4-5*, se puede observar como los *nodos Skeleton* contiene información suficiente para definir cualquier tipo de estructura articulada, y si bien, su representación gráfica no resulta tan intuitiva como una aproximación tradicional basada en parejas *Segmento-Junta*, la información presenta una estructura más compacta, y su integración en el *grafo de escena* resulta más coherente.

4.2.2 Representación de un actor sintético sobre una jerarquía de dibujado TR. Bases para la definición del Nodo Actor.

El primer objetivo en el proceso de representación de un actor sintético en *grafo de escena* es encontrar un método adecuado para definir su estructura articulada. Esto puede ser conseguido mediante la utilización de los *nodos Skeleton* presentados en el apartado anterior. Pero además, es necesario disponer de alguna estructura adicional que permita hacer que un actor sea algo más que un conjunto de *nodos Skeleton* dispersos por el *grafo de escena,* y que proporcione un aspecto encapsulado al fragmento de *grafo de escena* que representa a un actor virtual. El nodo *Actor* actúa como padre del resto de nodos que componen el actor, permite ubicar al actor en la escena, actúa como centro de control los nodos que componen el actor, y es el responsable de gestionar métodos de *culling* y gestión de *LOD* especialmente diseñados para la gestión de actores virtuales.

En *Figura 4-6* se puede observar una representación de un actor virtual muy sencillo definido a partir de un nodo *Actor* y seis *nodos Skeleton* correspondientes con los puntos de articulación de la cabeza, el tórax, los dos brazos y las dos piernas. Las celdas de color blanco representan los nodos que contienen la información poligonal de las distintas geometrías que componen el actor.

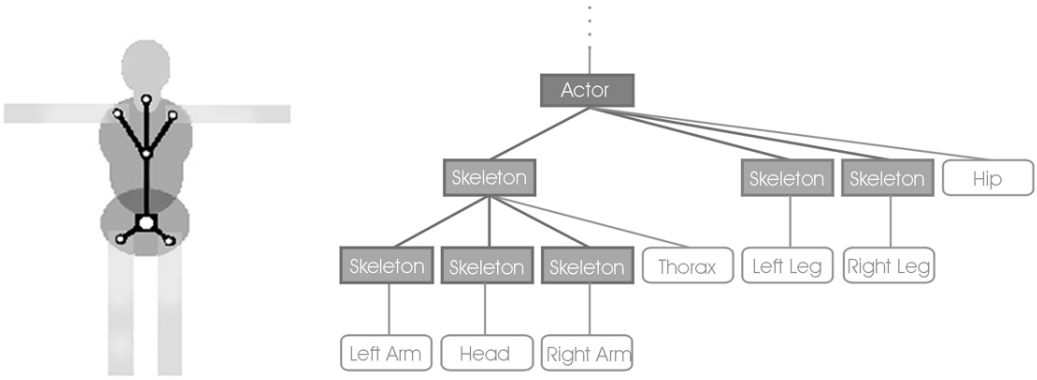


Figura 4-6. Representación de un actor muy sencillo y su correspondiente jerarquía de nodos *Actor* y *Skeleton*.

4.2.3 Integración en el funcionamiento estándar de un Grafo de Escena.

El objetivo de este trabajo no es definir un nuevo tipo de *grafo de escena*, sino proporcionar nuevas estructuras de datos y métodos, que añadidas sobre los existentes en los *grafos de escena* actuales, los hagan capaces de gestionar adecuadamente una escena que presente múltiples actores virtuales. En el estado del arte se ha realizado una descripción de los principios básicos de un *grafo de escena,* y también se ha descrito el funcionamiento de los nodos típicos que constituyen un *grafo de escena*. Para la implementación de un actor virtual sencillo bastaría con la existencia de nodos *Actor* y *Skeleton,* y algún tipo de nodo que almacenase información poligonal (de un modo similar al mostrado en la Figura 4-6). Sin embargo, este tipo de

integración resultaría muy limitado. Es adecuado que los nodos *Actor* y *Skeleton* desempeñen dentro del *grafo de escena* un papel similar a un nodo grupo tradicional, es decir, que puedan tener como hijos a nodos de cualquier tipo, y a su vez ser hijos de cualquier tipo de nodo. La única excepción a esta regla está va a dar en los nodos *Skeleton*, que siempre han de ser hijos de un nodo *Actor* o de otro nodo *Skeleton*.

El hecho de que los nodos *Actor* y *Skeleton* se comporten como nodos normales dentro del g*rafo de escena* permite establecer relaciones de dependencia jerárquica de cualquier tipo, posibilitando que un actor virtual pueda ser insertado en cualquier parte del *grafo de escena*, y también que un fragmento de *grafo de escena* pueda depender de un nodo *Actor* o de un nodo *Skeleton*. Así, ha de ser posible colocar a un actor virtual dependiendo de un nodo de tipo transformación afín con el cual se modifica la posición de un vehículo en la escena, y conseguir con ello que el actor virtual se desplace de forma solidaria con el vehículo, también ha de ser posible colocar un fragmento de *grafo de escena* complejo (por ejemplo uno que defina algún tipo de herramienta), dependiendo del nodo *Skeleton* que se corresponde con la articulación de la muñeca de un actor de tipo humanoide, y conseguir de este modo lo que la herramienta se mueva vinculada a la posición de su muñeca. Esto, además, permite actuar de forma adecuada a los métodos de culling standard, así, es posible hacer que varios actores virtuales sean agrupados debajo de un nodo de tipo grupo, y que un algoritmo tradicional de culling decida si el grupo completo de actores puede ser eliminado o no del proceso de dibujado, incluso si ese grupo de actores pasase a depender de un nodo que representa un edificio, se podría evitar automáticamente su procesado y dibujado en el caso de que el algoritmo de gestión de *culling* determinase que el edificio entero se encuentra fuera de la pirámide de visión.

En la *Figura 4-7* se puede apreciar un ejemplo muy sencillo de como un actor virtual puede presentar distintos tipos de interrelación con el resto del *grafo de escena,* y como se realiza el proceso de recorrido del *grafo de escena* y la acumulación de transformaciones. En la figura se representa una escena formada por el actor virtual humanoide de la *Figura 4-6*, que ha sido colocado en el interior de un automóvil, que se está desplazando por la escena, y en cuya muñeca, además, se ha colocado un reloj analógico con movimiento tanto en su aguja horaria como en su minutero. Los nodos etiquetados con el nombre de "Transform" representan a los nodos de transformación afín existentes en cualquier *grafo de escena* tradicional, estos nodos comparten con los nodos *Actor* y *Skeleton* dos características básicas: son capaces de comportarse como un grupo, es decir pueden tener varios hijos, y son capaces de aplicar una determinada matriz de transformación.

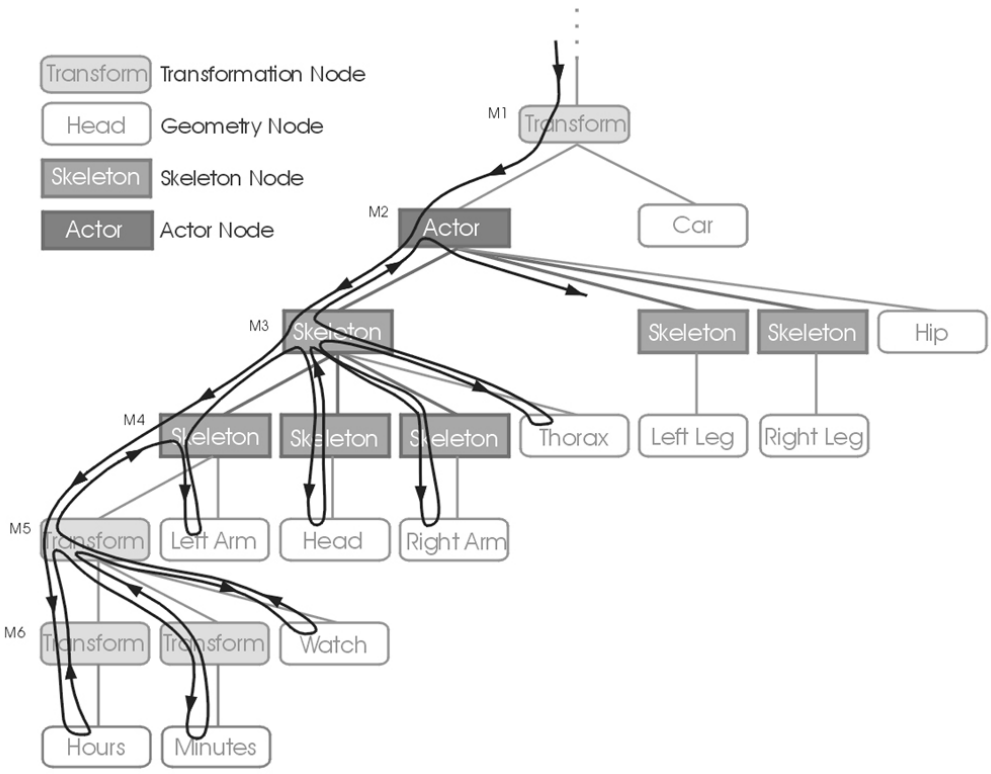


Figura 4-7. Integración de un actor virtual sencillo en una escena genérica mostrándose también el orden de recorrido de los nodos en un *grafo de escena*.

En la figura se representa mediante un trazo de color negro cual es proceso de recorrido del *grafo de escena* (en los textos en ingles denominan "*traversal"* a esta operación). Mediante las etiquetas M1, M2, M3, etc., se representan las matrices de transformación aplicadas por los nodos de *Transformación afín,* y también por los nodos *Actor* y *Skeleton*. Tal y como se muestra en la figura, se puede considerar que el procesado del *grafo de escena* consiste en realizar un recorrido secuencial a través de los nodos que la componen, existiendo una matriz de transformación que es modificada a medida que se avanza en el recorrido sobre el *grafo de escena*. Existe una pila de matrices en cuya cima siempre está la matriz actual, cuando se avanza hacia un nodo hijo que aplique una matriz de transformación, se hace un *push* sobre dicha pila, y en la parte alta de la pila se inserta una nueva matriz que es el resultado de multiplicar la matriz anterior por la matriz del nuevo nodo. Cuando en el proceso de recorrido del *grafo de escena* es necesario retornar hacia atrás, se producen sucesivos desapilados[1]. Un nodo es procesado teniendo en cuenta la matriz que se encuentre en la parte alta de esta pila, si, por ejemplo, el nodo a procesar fuese un nodo con información geométrica, sus vértices y normales serían multiplicados por dicha matriz en instantes previos a su dibujado, así los vértices del nodo de información geométrica etiquetado como "Hours" serían multiplicados por una matriz composición de M1*M2*M3*M4*M5*M6, el etiquetado como "Watch" por una matriz que sería M1*M2*M3*M4*M5, y el

[1]Esta pila de matrices es proporcionada por la OpenGl, su contenido es modificado mediante las operaciones *glLoadMatrix()*, *glMultMatrix()*, *glPushMatrix()*, *glPopMatrix()*, etc., y consultado mediante la orden *glGet(GL_MODELVIEW_MATRIX)*.

etiquetado como "Left Arm", por la matriz compuesta de M1*M2*M3*M4. El recorrido del *grafo de escena* es hecho a cada frame, de tal modo que si la matriz de algún nodo sufre alguna modificación, su efecto es propagado de forma automática a todo el fragmento de *grafo de escena* que depende directamente de dicho nodo.

Cuando en este trabajo se habla de nodos *Actor* y nodos *Skeleton*, hay que considerar que presentan una estructura básica similar a la de un nodo de tipo Grupo. La información particular del nodo *Skeleton* es almacenada en una estructura especial de nombre ***vaSkeleton***, la información particular de un nodo *Actor* es almacenada en una estructura de nombre ***vaActor***. Vistas desde un punto de vista muy simplificado, la misión de estas estructuras es retornar la matriz de transformación que los nodos *Actor* y *Skeleton* aplicarán al *grafo de escena*. En la Figura 4-8 se representa la relación existente entre los nodos *Actor* y *Skeleton* y las estructuras *vaSkeleton* y *vaActor*.

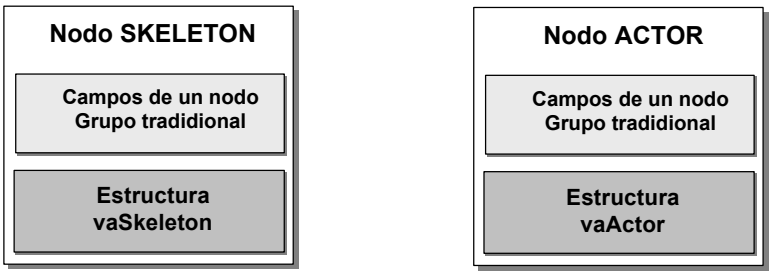


Figura 4-8. Nodos *Actor* y *Skeleton* implementados a partir como nodos de *Grafo de Escena*.

El contenido de las estructuras *vaSkeleton* y *vaActor* es analizado con detenimiento en los apartados que siguen, sin embargo, no se va a entrar a describir los campos que contienen estos nodos por el hecho de ser nodos grupo estándar de un *grafo de escena*. Estos campos vienen determinados por las características concretas de cada *grafo de escena*, y por tanto es una información que puede variar dependiendo del *grafo de escena* empleado como base.

Los dos nuevos tipos de nodos propuestos pueden ser aplicados en el diseño de algún nuevo tipo de *grafo de escena*, o bien ser integrados en alguno de los ya existentes. Para ello es necesario que el *grafo de escena* sobre el que se integre tenga cierta capacidad de extensión. En la actualidad existen dos métodos principales para personalizar el comportamiento de un nodo de un *grafo de escena*:

- En algunos *grafos de escena*, los nodos disponen de una zona preparada para almacenar información proporcionada por el usuario, y también permiten la especificación de rutinas de *callback* que serán llamadas automáticamente durante el "traversal" del *grafo de escena*. En este caso el nodo *Actor* sería un nodo de tipo *Grupo,* y la estructura *vaActor* estaría en la zona de datos de usuario de ese nodo, las rutinas de callback se encargarían de utilizar la información de la estructura *vaActor* para calcular y aplicar una matriz de transformación. Algo equivalente ocurriría con el nodo *Skeleton*.
- Algunos *grafos de escena* presentan una orientación a objetos, siendo los nodos del *grafo de escena* clases a partir de los cuales es posible generar subclases que permitan añadir nuevos campos y funciones a la estructura básica de nodo. En estos casos los nodos *Actor* y *Skeleton* podrian ser implementados como subclases de un nodo *Grupo* de ese *grafo de escena*.

4.3 Nodo *Skeleton*.

Un nodo *Skeleton* es un nuevo tipo de nodo, que además de tener las características de un nodo grupo, actúa como elemento modular único a partir del cual ha de ser posible definir cualquier tipo de estructura articulada natural o artificial. El nodo *Skeleton* está formado por tres tipos de informaciones distintas (ver Figura 4-9), que le hacen capaz de definir una estructura articulada sin necesidad de recurrir a la tradicional combinación *Segmento-Junta*:

- Un bloque de información sobre los *grados de libertad* existentes en esa articulación.
- Un *offset* que almacena la información referente a las diferencias que existen entre el sistema sistemas de referencia del propio nodo y el de su antecesor.
- Un elemento auxiliar que actúa como *interface* para seleccionar y modificar los valores existentes en los grados de libertad del nodo.

Figura 4-9. Elementos constitutivos de un Nodo *Skeleton*.

En los siguientes apartados se explican los criterios empleados para almacenar de forma adecuada la información referente a los grados de libertad, también como se almacena la información referente al offset con respecto a su nodo antecesor, y la forma en la que se ha definido la información sobre la interface para manipulación directa del actor. Una vez presentados los criterios empleados en la definición del contenido de un nodo *Skeleton*, se muestra su estructura de datos, y por último se analiza el coste computacional derivado de la gestión de los nodos *Skeleton,* y se proponen distintas estrategias que permiten maximizar su rendimiento.

4.3.1 Criterios para establecer la información referente a los Grados de Libertad.

La misión principal de un nodo *Skeleton* es almacenar la información sobre las transformaciones afines originadas como consecuencia de la modificación de alguno de sus grados de libertad. En la actualidad existen distintos sistemas de definición de estructuras articuladas que emplean diferentes criterios: algunos hacen que un nodo solamente pueda contener un grado de libertad, otros distinguen entre puntos de articulación rotacionales y traslacionales, otros tienen un variado conjunto de diferentes tipos de juntas definidas para problemas específicos. La variedad de soluciones dadas a la implementación de estructuras

articuladas es muy extensa, pero es común que se centren en la pura descripción topológica, en la comodidad de su interface, o en la solución algún problema específico, no teniendo una intención clara de estandarización, y no prestando especial atención a aspectos tales como la eficiencia en un sistema de tiempo real, o a la forma de integración en un *grafo de escena*.

Se pretende que el nodo *Skeleton* sea un elemento modular a partir del cual pueda ser definida cualquier tipo de estructura articulada. Esto fuerza a que la estructura de este elemento modular haya de ser analizada de forma minuciosa, especialmente si se observa desde el punto de vista de la estandarización. En los siguientes apartados se analizan en detalle todos criterios relacionados con la definición de los grados de libertad de un nodo *Skeleton*. En primer lugar se muestran cuales son los criterios de ejes, orientación y codificación de las matrices de transformación empleados, a continuación se analizan los métodos existentes para la codificación de un cambio de orientación, llegándose a la conclusión de que los *Angulos de Euler* resulta la opción más adecuada. En un apartado posterior se realiza un análisis más profundo sobre distintos aspectos relacionados con la utilización de los *Angulos de Euler*. A continuación se describen las ventajas derivadas de la concatenación de transformaciones, y en el último apartado, se concluye definiendo cuales serán los grados de libertad que constituirán un nodo *Skeleton,* y cual será su orden de aplicación.

4.3.1.1 Criterios de ejes, orientaciones y matrices.

La gran mayoría de aplicaciones de simulación emplean un sistema "right-handed" en el que el eje Z positivo actúa como eje vertical [2]. El sistema de coordenadas que será empleado en este trabajo es mostrado en Figura 4-10. En este sistema, si se considera como eje de rotación el eje X, la dirección positiva de rotación es la que va del eje Y al eje Z, si el eje de rotación fuese el eje Y, la dirección positiva sería la que va del eje Z al eje X y si fuese el eje Z, sería la que va del eje X al eje Y.

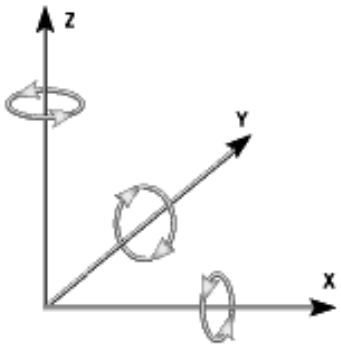


Figura 4-10. Sistema de Coordenadas. Giros especificados según la regla de la mano derecha.

[2] La utilización de sistemas "right-handed" esta ampliamente extendido en distintos ámbitos científicos. La utilización del eje Z como eje vertical se utiliza de forma habitual en el mundo de la simulación. En algunos sistemas antiguos basados en el estándar *SAE J670*e [SAE76] emplean un eje vertical Z negativo, los estándares más modernos [ISO91] proponen la utilización del eje Z positivo.

Otro de los aspectos que es necesario fijar es la forma en la que son definidos los puntos en el espacio 3D, lo cual afecta a la forma en la que se definen las matrices de transformación, y también a su orden de concatenación.

En la mayoría de textos que describen de forma matemática las transformaciones mediante matrices, se emplea una nomenclatura en la que los puntos tridimensionales son tratados como vectores columna $v = \begin{pmatrix} v_x & v_y & v_z & 1 \end{pmatrix}^T$, esta consideración afecta a la forma en la que se codifican las matrices de transformación (*column-major*), haciendo entre otras cosas las matrices de traslación almacenen su información sobre los elementos de la última columna, y también a la forma en la que se concatenan (post-multiplicacion). La utilización de una representación en la que los puntos están definidos como vectores fila $v = \begin{pmatrix} v_x & v_y & v_z & 1 \end{pmatrix}$ afecta a las matrices de transformación (*row-major*), haciendo que, por ejemplo, una matriz de traslación almacene sus valores en la última fila, y forzando a que las matrices se concatenen mediante pre-multiplicación. En la *Figura 4-11* se muestra la forma en la que se aplica una traslación y una rotación a un punto según ambos criterios, se puede observar como la elección de un determinado criterio para la representación de los vértices afecta a la forma que presentan las matrices de transformación, y tambien al orden en el que se han de realizar las operaciones.

Notación basada en vectores columna.

$$\underbrace{\begin{bmatrix} V'x \\ V'y \\ V'z \\ 1 \end{bmatrix}}_{\mathbf{v'}} = \underbrace{\begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\textbf{Translación}} \times \underbrace{\begin{bmatrix} \cos Rz & -sinRz & 0 & 0 \\ sinRz & \cos Rz & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\textbf{Rotación eje z}} \times \underbrace{\begin{bmatrix} Vx \\ Vy \\ Vz \\ 1 \end{bmatrix}}_{\mathbf{v}}$$

Notación basada en vectores fila.

$$\underbrace{\begin{bmatrix} V'x & V'y & V'z & 1 \end{bmatrix}}_{\mathbf{v'}} = \underbrace{\begin{bmatrix} Vx & Vy & Vz & 1 \end{bmatrix}}_{\mathbf{v}} \times \underbrace{\begin{bmatrix} \cos Rz & sinRz & 0 & 0 \\ -sinRz & \cos Rz & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\textbf{Rotación eje z}} \times \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Tx & Ty & Tz & 1 \end{bmatrix}}_{\textbf{Translación}}$$

Figura 4-11. Distintos tipos de notación empleados para representar transformaciones.

Si se sigue la notación basada en vectores fila, la representación de las matrices coincide con la representación empleada por los lenguajes de programaciónmás ampliamente extendidos (c, c++, java, pascal, python,...), ésta y otras razones han hecho que en el mundo de los gráficos por computadora se haya empleado a veces esta nomenclatura.

En este trabajo se empleará una notación basada en ***vectores columna*** (y por tanto matrices row-major y concatencaciones por postmultipliación), que a pesar de presentar ciertos inconvenientes desde el punto de

vista de la programación, coincide con la nomenclatura empleada en otras áreas científicas, y, además, está siendo adoptada por mayoría de textos sobre informática gráfica actuales.

4.3.1.2 Elección de método de representación de las orientaciones en un nodo *Skeleton*.

En la actualidad existen distintos métodos para definir un cambio de orientación en el espacio tridimensional. Los más importantes son las *Matrices de Transformación*, los *ángulos de Euler*, y los *Quaternions*. Cada uno de estos sistemas presenta determinadas ventajas y determinados inconvenientes, existiendo métodos para realizar traducciones entre ellos.

En este los siguientes apartados se describen la capacidad de codificación de orientaciones de las *Matrices 4X4*, los *Angulos de Euler* y los *Quaternios.* En un apartado final se llega a conclusiones sobre la forma en la que se codificaran las orientaciones en un nodo *Skeleton*.

4.3.1.2.1 Matrices de 4x4.

Mediante una matriz de 3x3 números flotantes es posible codificar un cambio de orientación en el espacio, en el caso de emplear matrices 4x4, es, además, posible codificar traslaciones y otro tipo de transformaciones muy útiles en la representación de un espacio tridimensional. Mediante la concatenación de matrices 4x4 es posible definir cualquier transformación espacial basada en traslaciones, rotaciones y escalados, siendo posible que una única matriz de 4x4 almacene una composición muy compleja de transformaciones. La mayoría de librerías gráficas de bajo nivel (incluida la OpenGl), y tarjetas gráficas existentes en la actualidad esperan ser alimentadas con este tipo de matrices. Esto último fuerza a que cualquier transformación espacial se haya presentar su resultado final en forma de una matriz 4x4.

La codificación de una orientación dentro de una matriz de transformación presenta características matemáticas muy adecuadas, pero su manejo resulta muy poco intuitivo: El valor de la orientación es almacenado en un conjunto de 9 valores dentro de la matriz. Intentar comprender cual es la orientación a partir de la simple observación de la matriz es una tarea muy dificultosa.

4.3.1.2.2 Angulos de Euler.

La forma más sencilla de describir una orientación en el espacio fue descubierta hace varios siglos por el matemático suizo Euler. En su "Teorema de la Rotación" asegura que una rotación arbitraria puede ser parametrizada empleando tan sólo tres variables, las cuales son conocidos comúnmente como *Angulos de Euler*. Según este teorema, cualquier orientación espacial puede ser descompuesta en una serie de tres rotaciones sobre los ejes principales.

La descripción de una orientación espacial a partir de *Angulos de Euler* es utilizada ampliamente en campos científicos tales como la Matemática, la Física y la Biomecánica, es el sistema que define una orientación arbitraria con el menor número de parámetros (tan sólo 3 frente a los 9 necesitados por una matriz de transformación, o lo 4 requeridos por un *Quaternion*), y, además, resulta muy intuitivo.

En este sistema se aplican de forma consecutiva tres rotaciones sobre los ejes principales, ocurriendo que el eje de rotación actual dependa del resultado de las rotaciones previas, por tanto la orientación final alcanzada depende del orden en el que sean aplicadas las tres rotaciones. Esto hace que sea necesario definir el orden de aplicación de los ángulos. Este sistema presenta un par de inconvenientes: El primero de ellos es conocido con el nombre de "*Gimbal-Lock", y* es un problema de tipo práctico que se encuentran los animadores cuando intentan colocar un objeto en una orientación arbitraria, y el segundo aparece cuando se quiere realizar una interpolación lineal entre dos orientaciones distintas descritas mediante *ángulos de Euler*. Ambos problemas y su solución son tratados en detalle más adelante.

4.3.1.2.3 Quaternions.

Los *Quaternions* permiten describir un cambio de orientación genérico a partir de un conjunto de 4 parámetros. Los *Quaternions* fueron inventados por el matemático William Rowan Hamilton (1809-1863) en 1843 cuando intentaba generalizar los números complejos en el espacio tridimensional. Posteriormente Shoemake [SHO85][SHO87] analizo las propiedades de los quaternions unidad para representar rotaciones en tres dimensiones. Los quaternios son números formados por una parte real y tres partes imaginarias y poseen su propia álgebra y propiedades. Los quaternions unidad tienen ciertas características interesantes como el hecho de que las multiplicaciones entre ellos sean muy rápidas, lo cual permite concatenar rotaciones de forma mucho más rápida que con matrices, o el hecho de que su parametrización no genere singularidades como el gimbal-lock. Adicionalmente solucionan el problema de la interpolación entre rotaciones de forma muy elegante y rápida a través de SLERP (Spherical Linear Interpolation)[WAT92].

Todas estas características han hecho que hayan considerado como una alternativa ventajosa respecto al tradicional empleo de matrices de transformación, sin embargo, como veremos, su aplicación sobre estructuras articuladas solamente resulta útil en circunstancias muy concretos.

4.3.1.2.4 Conclusiones.

El hecho de que las librerías gráficas de bajo nivel y tarjetas gráficas existentes estén preparadas para trabajar con matrices de 4x4, así como el hecho de que sea el único sistema capaz de codificar también translaciones, o incluso proyecciones fuerza a que cualquier transformación espacial haya de presentar su resultado final en forma de una matriz 4x4.

El principal inconveniente de la representación matricial de una orientación es que para la codificación de un cambio de orientación genérico son necesarios 9 parámetros que además resultan poco intuitivos. Este inconveniente puede ser subsanado empleando una codificación de la orientación basada en *Angulos de Euler*, la cual puede ser traducida a una representación matricial. Es un sistema que sólo requiere 3 parámetros, resulta intuitivo y está muy extendido.

Respecto a los quaternions, presentan dos ventajas respecto a la codificación mediante Euler y matrices, la primera de ellas es que se concatenan más rápido que las matrices, y la segunda que la interpolación entre dos orientaciones es realizada de forma optima. Sin embargo, a pesar de estas características, la utilización de

quaternions no ha de ser considerada como una alternativa a la utilización de matrices de transformación y a la utilización de *ángulos de Euler*, puesto que su representación de las orientaciones resulta muy poco intuitiva en comparación con los *ángulos de Euler*, y el hecho de solamente se puedan emplear para parametrizar rotaciones hace que no sean comparables con las matrices de transformación.

El único caso en el que los *Quaternios* pueden ser útiles dentro de un nodo *Skeleton,* es como herramienta de calculo intermedio para realizar la interpolación entre dos orientaciones espaciales distintas. En este caso la función SLERP de los *quaternions,* presentan una solución óptima, para emplearlo, se pueden traducir las dos orientaciones Euler a una representación en forma de quaternion, realizar la interpolación entre los *quaternions* y traducir el *quaternion* fruto de la interpolación a una matriz de 4x4 que es la que finalmente se aplica.

4.3.1.3 Consideraciones relacionadas con los *Angulos de Euler*.

La codificación de la orientación mediante *ángulos de Euler* emplea un número mínimo de parámetros además de ser intuitiva y está muy extendida. Sin embargo, como se ha citado anteriormente, presenta algunos inconvenientes que han de ser considerados: Es necesario fijar un orden de aplicación de las rotaciones, presentan el problema del "Gimbal-Lock", y una forma incorrecta de interpolación angular. Estos aspectos son analizados con detenimiento en los siguientes apartados.

4.3.1.3.1 Selección de un orden de aplicación de los ángulos de Euler.

Uno de los problemas que presentan los *ángulos de Euler* consiste en elección de los ejes sobre los que se aplican las tres rotaciones y el orden de aplicación. En la actualidad no existe un estándar que defina cual es el criterio más adecuado. Quizá ha sido el área de la biomecánica la que ha hecho un esfuerzo más importante para establecer un sistema estándar de *ángulos de Euler*, a pesar de ello, en algunos documentos actuales aún se sigue indicando que la mejor alternativa es caracterizar cada una de las articulaciones empleando un sistema específico de ejes.

En la realidad existen 12 posibilidades válidas para definir *ángulos de Euler* (XYX, XYZ, XZX, XZY, YXY, YXZ, YZX, YZY, ZXY, ZXZ, ZYX y ZYZ), y ningún criterio que haga que un sistema sea claramente mejor que el resto. Sin embargo, desde el punto de vista de la estandarización buscada por el nodo *Skeleton*, es totalmente necesario establecer un criterio fijo que determine cuales son los ejes sobre los que se aplican las transformaciones, y en que orden, en este sentido existe un orden de aplicación de las transformaciones que se utiliza de una forma estandarizada en campos como la informática gráfica, aviación, náutica o en sistemas de sensorización, este orden es conocido como *"heading-pitch-roll"* y que se basa en la aplicación de tres giros consecutivos sobre tres ejes a los que se suele denominar eje vertical, eje lateral y eje longitudinal. El ejemplo típico para imaginarse este sistema de codificación de la orientación consiste en suponer un aeroplano colocado con la cabina orientada hacia el extremo positivo del eje longitudinal y las alas con una orientación paralela al eje lateral. Según este criterio, el aeroplano puede ser colocado en cualquier orientación por medio de las siguientes transformaciones:

- Aplicar un valor de ***"heading"*** (también denominado *"yaw"*) sobre el eje vertical. Esto modificaría la orientación del aeroplano a derecha o a izquierda.
- Aplicar un valor de "**pitch**" sobre el eje lateral. Con esto se orientaría al aeroplano hacia abajo o hacia arriba.
- Aplicar un valor de ***"roll"*** sobre el eje longitudinal. Con esto se conseguiría que una de las alas tomase una posición más elevada que la otra.

Siguiendo el criterio de ejes establecido en el apartado 4.3.1.1, el eje vertical es el eje Z positivo, como eje longitudinal se empleará el eje X positivo, puesto que a pesar de que librerías como OpenGl Performer emplean el eje Y como eje longitudinal, la utilización del eje X como eje longitudinal forma parte de los estándares empleados en la simulación de vehículos [SAE76] [ISO91]. De forma consecuente el eje lateral será el eje Y positivo. En la *Figura 4-12* se puede muestran cuales son los *ángulos de Euler* empleados para controlar la orientación de una parte de un actor.

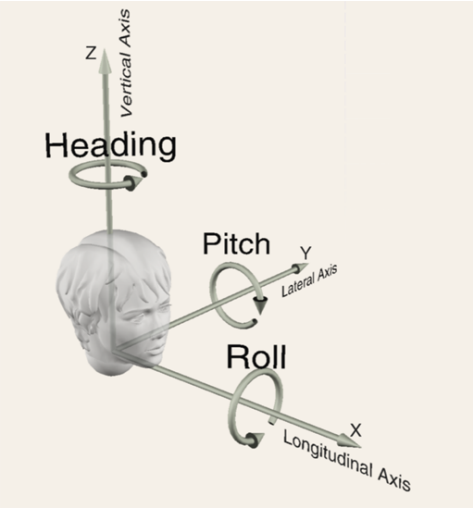


Figura 4-12. Sistema de ejes y *ángulos de Euler* empleados en la especificación de la orientación de un nodo *Skeleton*.

Los *Angulos de Euler* a aplicar en los nodos *Skeleton* se aplicaran de forma consecutiva siguiendo un orden **Z®Y®X** del siguiente modo: Primero se aplica el valor de *"Heading"* en torno al eje Z positivo, eso hace que los ejes X y Y pasen a tener unas nuevas orientaciones. A continuación se aplica el valor de *"Pitch"* sobre el nuevo eje Y positivo, esto hace que el eje Z pase a estar en una segunda orientación y el eje X en una tercera orientación, y por último se aplica el valor de *"Roll"* sobre el nuevo eje X positivo.

4.3.1.3.2 Gimbal-Lock.

El hecho de que al aplicar una rotación de Euler estemos modificando los ejes relativos sobre los que se va a aplicar la siguiente tiene como inconveniente que a veces ocurra que uno de los ejes de rotación sea mapeado sobre otro eje de rotación. El resultado de esta coincidencia sería la perdida de un grado de libertad. Para entenderlo mejor, supongamos que en el ejemplo del avión ya tratado aplicamos un determinado valor de rotación R1 sobre el eje Z (*Heading*), a continuación aplicamos una rotación de 90 grados sobre el eje Y (*Pich*), y a continuación un valor de rotación R3 sobre el eje X (*Roll*). En el caso de aplicar las transformaciones en este orden, ocurre que la rotación R3 sobre el eje X tiene el mismo efecto que la rotación inicial R1 en torno al eje Z. La rotación de 90 grados sobre el eje Y ocasiona que los ejes X y Z estén alineados, lo que supone la pérdida de un grado de libertad. En [WAT92] se describe como la parametrización basada en *ángulos de Euler* presenta este tipo de singularidades indicando que es un problema inherente a la utilización de *ángulos de Euler* y que es independiente del criterio empleado en la selección de los ejes de rotación, y su orden de aplicación.

La codificación mediante *ángulos de Euler* es adecuada puesto que cualquier orientación espacial puede ser conseguida mediante una combinación de valores de Euler, el problema está en que si se diseña una interfaz de usuario en el que se tiene acceso directo a estos 3 parámetros, la interdependencia entre ellos hace necesario que el animador conozca lo que es el *"gimbal-lock"* e intente evitarlo. Una solución consistiría en enmascarar los valores de Euler al animador y presentarle un tipo de interfaz más intuitivo que tenga la capacidad de traducir internamente a valores de Euler.

4.3.1.3.3 Interpolación angular.

Otro problema derivado de la parametrización de la orientación mediante *ángulos de Euler* reside en el hecho de que la interpolación lineal entre dos orientaciones distintas, no presenta una solución única, y resulta impredecible desde un punto de vista práctico. Este problema deriva del hecho de que al realizar una interpolación lineal entre los *ángulos de Euler,* estamos tratándolos como si fueran independientes entre sí, cuando en realidad existe un orden de aplicación de las transformaciones que no es conmutativo. El *"teorema de Euler"* indica que es posible cambiar de una orientación espacial 3D a otra distinta empleando una única rotación en torno a un único eje, ésta es desde luego la forma de interpolación ideal entre dos orientaciones distintas. Sin embargo, empleando una parametrización basada en *ángulos de Euler* no es posible realizar este tipo de interpolación. Afortunadamente, la parametrización mediante *quaternions*, si bien resulta matemáticamente más compleja, proporciona el tipo de interpolación que estamos buscando, y puede ser empleada de forma auxiliar.

4.3.1.4 Concatenación de transformaciones.

El hecho de que los *ángulos de Euler* se apliquen de una forma consecutiva, hace que las primeras transformaciones afecten a los ejes de los sistemas de coordenadas en los que se definen las transformaciones siguientes. Varias matrices de transformación consecutivas pueden ser multiplicadas y sustituidas por una única matriz que realiza la operación completa, como veremos, esto supone un incremento de rapidez considerable.

La aplicación de una secuencia consecutiva de transformaciones a un conjunto de vértices es un proceso que se repite continuamente durante el dibujado de un actor virtual. Existen dos formas diferentes para realizar esta operación, considerar que las transformaciones se aplican de forma independiente entre sí, o bien considerar que las transformaciones se aplican de forma concatenada. Veamos sus diferencias mediante un ejemplo:

Supongamos el caso de una articulación con dos grados de libertad, cada uno de los cuales es representado mediante una matriz de transformación (denominaremos ***M1*** y ***M2*** a las matrices de cada uno de los grados de libertad). Supongamos que queremos utilizar ese par de matrices para trasformar el vértice ***v*** y conseguir un vértice transformado ***v”*.**

En el caso de transformaciones independientes:

v’ = M1* v **v” = M2* v’** **v” = M2* (M1 * v)**

En el caso de transformaciones concatenadas:

M’= M2 * M1 **v” = M’ *v** **v” = (M2* M1) * v**

La concatenación de matrices permite que se puedan multiplicar dos matrices para obtener una tercera, y ésta tendrá el mismo efecto en los puntos tridimensionales que la aplicación consecutiva de las dos matrices originales. Por ejemplo, si tiene una matriz que rota un punto, y otra que lo mueve en una dirección, se puede combinarlas para producir una matriz que realizara la rotación y translación en un único paso. Se pueden crear transformaciones extremadamente complejas de este modo, consiguiendo que cada punto 3D a ser representado sólo tenga que ser multiplicado por una única matriz.

Si tan sólo se pretende transformar un único vértice el coste es equivalente en ambos casos. La diferencia aparece cuando el número de vértices a transformar es elevado, puesto que el cálculo de la matriz concatenada (**M’**) tan sólo se ha de realizar una vez.

Las mejora en rapidez inherente a la utilización de transformaciones concatenadas hace que las librerías gráficas de bajo nivel como la OpenGl estén especialmente preparadas para su utilización. La mayoría de *grafos de escena* comerciales, e incluso las tarjetas gráficas presentan características orientadas a la utilización de este tipo de transformaciones.

4.3.1.5 Selección de transformaciones y orden de aplicación.

La elección del número de grados de libertad que componen un nodo *Skeleton*, cuales han de ser estos grados de libertad, y el orden en el que se han de aplicar es uno de los aspectos que han de ser tratados con mayor delicadeza. Distintos tipos de aplicación que han abordado el problema de la implementación de una estructura articulada han empleado distintas aproximaciones, yendo desde la utilización de estructuras elementales que se pueden agrupar para definir articulaciones complejas, hasta el empleo de estructuras específicamente diseñadas para representar algún tipo particular de articulación.

Cualquier articulación compleja pueda ser definida a partir de varias estructuras de tipo junta sencillas que se apliquen de forma consecutiva. Existen *grafos de escena* como el utilizado por VRML en los que los nodos empleados para definir los puntos de articulación (nodos Transform en este caso) tan sólo tienen un grado de libertad, y las articulaciones más complejas han de ser definidas enlazando de forma consecutiva varios de estos nodos. Para el diseño de los nodos *Skeleton* podrían seguirse dos políticas distintas: Definir un nodo *Skeleton* que contenga una gran cantidad de grados de libertad que permita definir cualquier tipo de punto de articulación, o bien, emplear un conjunto de nodos *Skeleton* elementales que puedan ser agrupados el caso de que sea necesario definir articulaciones complejas. La principal ventaja de la segunda alternativa es que permite que los grados de libertad se apliquen en cualquier orden, sin embargo hace que el *grafo de escena* resulte más complejo, además, conceptualmente resulta mucho más adecuado que todas las transformaciones referentes a un punto de articulación, sean incluidas en una única estructura. La primera alternativa consigue esto último, pero como contrapartida hace que el nodo *Skeleton* presenten una estructura interna más compleja, ya que existen muchas posibles variantes respecto al número de grados de libertad, y el orden en el que se aplican (especialmente en el caso de estructuras articuladas artificiales). Se va a buscar una solución de compromiso entre estas dos alternativas recurriendo a los siguientes criterios:

- Siguiendo en la línea de hacer que la representación del *grafo de escena* sea lo más sencilla posible, se va a intentar evitar que para la definición de un único punto de articulación, sea necesaria la utilización de varios nodos.
- Se va a dar prioridad a la implementación de puntos de articulación pertenecientes a formas orgánicas. En general, se puede decir que cualquier punto de articulación de este tipo puede ser definido con en base a tres grados de libertad rotacionales.
- Es necesario que un nodo *Skeleton* tenga capacidad para definir estructuras articuladas artificiales simples. En el caso de que presenten una complejidad especial han de poder ser definidas a partir de la aplicación de varios nodos *Skeleton* consecutivos.
- La eficiencia a la hora de la ejecución es primordial, y eso fuerza a que las matrices que definen los grados de libertad se apliquen de forma concatenada, y que, por tanto, exista un orden prefijado en la aplicación de los grados de libertad.

Para que un nodo *Skeleton* pueda definir cualquier punto de articulación orgánico, ha de tener tres grados de libertad rotacionales, para que además pueda servir para implementar articulaciones artificiales, ha de ser capaz de aplicar traslaciones. Para conseguir que el nodo *Skeleton* sea lo más compacto posible, llevará un

único grado de libertad de tipo translacional. En los pocos casos en los que sea necesaria la descripción de articulaciones artificiales de mayor complejidad se recurrirá a la aplicación de varios nodos consecutivos. Un nodo *Skeleton* estará formado por tres grados de libertad de tipo rotacional y uno de tipo traslacional.

En los apartados anteriores se ha detallado como los *ángulos de Euler* son la forma más adecuada para que un nodo *Skeleton* represente un cambio de orientación en el espacio, y se ha seleccionado un criterio según el cual las tres rotaciones son aplicadas de forma concatenada sobre sus ejes locales Z, Y y X. Según este criterio, la rotación sobre el eje X es la aplicada en último lugar, resultando adecuado que actúe como eje longitudinal del segmento corporal sobre el que actúa. La forma coherente de añadir el único grado de libertad traslacional será aplicarlo también sobre el eje X, y concatenado a continuación de los grados de libertad rotacionales.

Según todo lo anterior, el nodo *Skeleton* estará formado por cuatro grados de libertad y su orden de aplicación será el siguiente:

$$\boxed{R_z \Rightarrow R_y \Rightarrow R_x \Rightarrow T_x}$$

A pesar de que un nodo *Skeleton* tenga 4 grados de libertad, en la mayoría de los casos tan sólo se emplean uno o dos, el nodo *Skeleton* dispondrá de mecanismos para indicar los grados de libertad que se estan empleando en cada situación.

4.3.2 Criterios para establecer la información referente al Offset.

Una de las funciones de un nodo *Skeleton* es almacenar la diferencia entre el sistema de referencia de un punto de articulación y el del nodo del que depende jerárquicamente. La diferencia entre estos sistemas de referencia consiste en un cambio de posición y también en un cambio de orientación, estas dos diferencias son almacenadas por separado bajo los nombres de *Offset de Traslación* y *Offset de Rotación* (ver Figura 4-13).

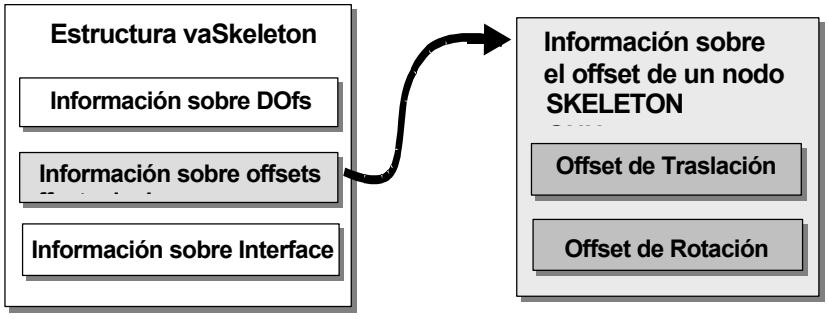


Figura 4-13. Elementos constitutivos del Offset de un Nodo *Skeleton*.

El ***Offset de Traslación*** es un vector que almacena la traslación que sufre el origen del sistema de coordenada de un nodo *Skeleton* con respecto al origen del sistema de coordenadas del nodo que le antecede.

La orientación del sistema de referencia de algunas articulaciones puede ser la misma que la de su padre, ocurre esto en el caso de la articulación de la muñeca respecto a la del codo, o en la del codo respecto a la del hombro. Sin embargo, es común que las articulaciones presenten diferentes orientaciones, un caso muy claro es la articulación del tobillo, la cual presenta una orientación distinta a la de la rodilla (ver Figura 4-14). El hecho de que los grados de libertad se apliquen mediante matrices concatenadas, hace necesario que el sistema de coordenadas sobre el que se aplican las concatenaciones tenga una orientación inicial adecuada y por tanto es necesario que esté orientado de forma personalizada para cada una de las articulaciones. Al conjunto de transformaciones que adecua la orientación del sistema de coordenadas de un punto de articulación, se le denomina ***Offset de Rotación,*** y es almacenado dentro del nodo *Skeleton* en forma de un vector.

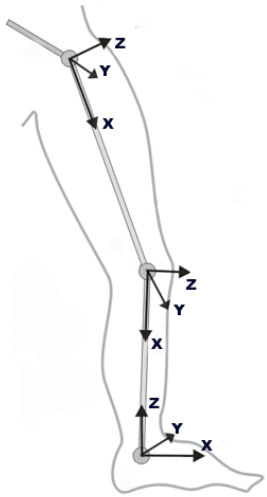


Figura 4-14. Diferencias entre los sistemas de referencia iniciales de distintas articulaciones.

El *Offset de Rotación* ha de ser capaz de transformar totalmente la orientación de un sistema de coordenadas, este objetivo puede ser alcanzado mediante la aplicación concatenada de tres matrices de rotación, una para cada uno de los ejes. El *offset* de un nodo *Skeleton* se define ha partir de un total cuatro matrices de transformación: una de traslación para la definición del *offset de traslación,* y tres de rotación para la definición del *offset de rotación*. El orden de aplicación de dichas matrices es el siguiente:

$$T_{xyz} \Rightarrow R_z \Rightarrow R_y \Rightarrow R_x$$

El hecho de que el offset de un nodo *Skeleton* permanezca invariable una vez definido, permite que estas 4 matrices puedan ser concatenadas en una única matriz.

4.3.3 Criterios para definir la interfaz 3D de un nodo Skeleton.

La tercera misión de un nodo *Skeleton* es proporcionar una interfaz tridimensional mediante el cual, el usuario puede observar y manipular los grados de libertad de la estructura esquelética del actor.

En la Figura 4-15 se puede observar una representación de la interface 3D para el manejo de los actores virtuales, se puede ver como el punto de articulación de cada nodo *Skeleton* es representado por una pequeña esfera (etiquetada como *Skeleton Center*), de igual modo, se puede observar como vinculados a cada punto de articulación hay unos cubos y pirámides etiquetados bajo el nombre de ***handlers*** y cuya utilidad es proporcionar acceso a una serie de elementos de interfaz que permiten modificar los valores de los grados de libertad. Los cubos etiquetados como *DK Handler* permiten acceder directamente a los grados de libertad de un nodo *Skeleton,* y modificarlos de forma interactiva. Las pirámides (nombradas como *IK handler*) permiten modificar la posición de las articulaciones empleando cinemática inversa. Cuando alguno de esos *handlers* es seleccionado, aparecen unos elementos de interfaz auxiliares en forma de flecha sobre los cuales es posible actuar y modificar los valores de los grados de libertad. En la figura se muestra un ejemplo en el que el usuario a pulsado sobre el *DK handler* de la articulación del codo, y con ello ha tenido acceso una flecha tridimensional (etiquetada como *DK arrow*) que indica que esa articulación solamente tiene un grado de libertad y permite su manipulación directa.

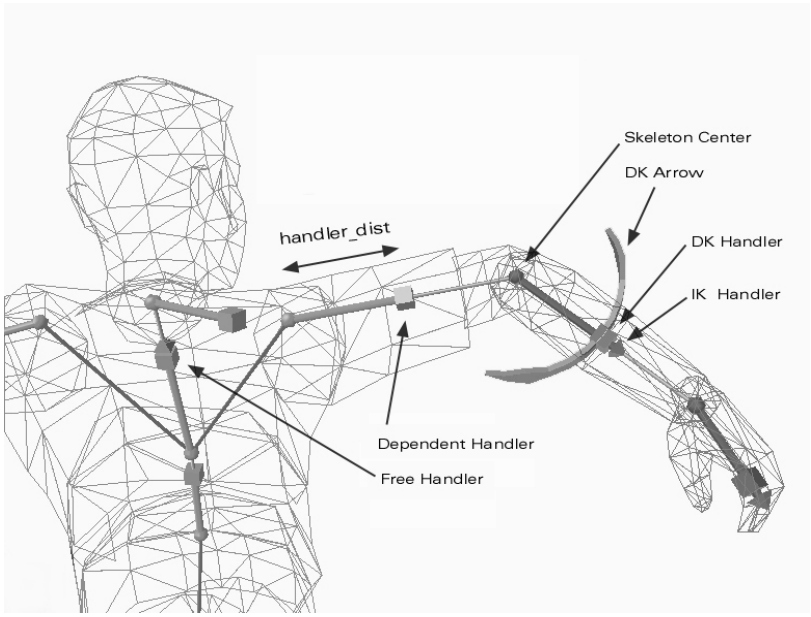


Figura 4-15. Elementos de la interfaz 3D para manipulación directa de un actor virtual.

En la figura también se puede observar como existen dos tipos de handlers (almacenados en el campo ***handlerType*** del nodo *Skeleton*): *Handlers* de tipo *Dependent,* que se utilizan en aquellas articulaciones que presentan tan sólo un hijo (tal es el caso de la articulación del hombro o del codo de un actor humano), en estos caso la posición del manipulador se modifica automáticamente para que aparezca alineado con la siguiente articulación, esto facilita la tarea de creación interactiva del esqueleto del actor, y *handlers* de tipo *Free* que se utilizan para actuar sobre los puntos de articulación que tienen más de un hijo, tal es el caso del punto de articulación de la columna vertebral mostrado en la figura del cual dependen los puntos de articulación de la cabeza y los hombros. La posición de estos *handlers* viene definida por el offset de rotación de dicho punto de articulación. La distancia a la que se encuentra el manipulador del centro punto de articulación es almacenada en un campo del nodo *Skeleton* (***handlerDist)***, este valor es también empleado para definir el radio de las flechas en forma de arco que se utilizan para realizar el control mediante

cinemática directa (ver la flecha etiquetada como "*DK Arrow*" en la Figura 4-15), y permite personalizar la interfaz de manejo de cada articulación.

4.3.4 Estructura de datos.

En los apartados anteriores se ha realizado un análisis sobre la información que debería ser almacenada un nodo *Skeleton* para poder ser capaz de describir la estructura articulada de un actor virtual. En este momento, ya se está en condiciones de detallar cual será la estructura de datos que almacenara la información de un nodo *Skeleton*:

```
typedef struct vaSkeletonStruct{
   float    offsetPos[3];       // Offset de traslacion.
   float    offsetHpr[3];       // Offset de rotacion. Orden rz->ry->rx.
   float    offsetMat[16];      // Matriz compuesta de offsetPos y offsetHpr.

   float    *dofRz;             // Grados de libertad.
   float    *dofRy;
   float    *dofRx;
   float    *dofTx;
   float    dofsMat[16];        // Matriz de grados de libertad: dofRz->dofRy->dofRx->dofTx.
   float    sklMat[16];         // Matriz total aplicada por el nodo esqueleto: offsetMat->dofsMat.

   float    sklAbsMat[16];      // Matriz absoluta del nodo Skeleton.

   int      dofsCfg;            // Configuracion de los dofs.
   int      dirty;              // Flag que indica si sklMat es valida.
   int      trackSkeleton;      // Flag que indica necesidad de conocer su matriz absoluta.

   float    handlerDist;        // Distancia de los handlers respecto al pto de articulacion.
   int      handlerType;        // Tipo de handler. Free o Dependent.

   void *   sgNode;             // Puntero a su nodo.

   vaActor *   act;             // Puntero a su vaActor.
}vaSkeleton;
```

En primer lugar se observan los campos ***offsetPos*** y ***offsetHpr***, que son los responsables de almacenar la información referente a los offsets de rotación y traslación. Los valores son almacenados en sendos vectores de tres floats, *offsetPos* almacena directamente los valores de traslación sobre los tres ejes, *offsetHpr* almacena los valores de rotación sobre los ejes X, Y y Z. A partir de los valores de *offsetPos* y *offsetHpr* es posible generar una matriz de transformación cuyo resultado es almacenado en el campo ***offsetMat***.

La información referente a los grados de libertad es almacenada en los campos ***dofRz***, ***dofRy***, ***dofRx*** y ***dofTx***. Sería lógico que estos campos almacenasen directamente valores *float*, pero se puede observar que en realidad son punteros, esto presenta dos utilidades, la primera de ellas es que se puede emplear para indicar si ese grado de libertad existe o no (no existirá en el caso de que el puntero contenga un valor NULL), la segunda es que permite que los valores de los grados de libertad estén almacenados en un lugar distinto al nodo *Skeleton*, esto será utilizado para establecer una de las relaciones básicas entre los nodos *Skeleton* y *Actor* que serán tratadas en detalle en el apartado 4.4.3. Las transformaciones aplicadas por los grados de libertad son concatenadas y almacenadas en la matriz ***dofsMat***. El campo ***sklMat*** contiene la matriz relativa

aplicada por el nodo *Skeleton*, y es generada a partir de la multiplicación de *offsetMat* y *dofsMat*. En el campo ***sklAbsMat*** se almacena una matriz absoluta que define la posición del nodo *Skeleton* respecto al sistema de coordenadas del mundo.

Los campos ***dofsCfg, dirty*** y ***trackSkeleton*** son empleados para reducir el coste computacional asociado a la generación de las matrices de transformación de los nodos *Skeleton,* su forma de utilización es explicada más adelante. Los campos ***handlerDist*** y ***handlerType*** sirven para almacenar el tipo y la distancia a la que se encuentran los handlers para manipulación de un *Skeleton*. El campo ***sgNode*** almacena un puntero al nodo del grafo de escena vinculado con la estructura *vaSkeleton*, y, por último, el campo ***act*** almacena un puntero al nodo actor al que pertenece este nodo *Skeleton*.

4.3.5 Procesado básico de un nodo Skeleton.

La generación de la matriz a aplicar por un nodo *Skeleton* depende de 10 parámetros (3 valores del *offsetPos*, otros 3 valores del *offsetHpr* y los 4 grados de libertad). La obtención directa de la matriz aplicada por un nodo *Skeleton* requeriría de la concatenación de las siguientes matrices:

$$\underset{matOffsetPos}{\begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 1 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}} \times \underset{matOffsetRz}{\begin{bmatrix} \cos Rz & -sinRz & 0 & 0 \\ sinRz & \cos Rz & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} \times \underset{matOffsetRy}{\begin{bmatrix} \cos Ry & 0 & sinRy & 0 \\ 0 & 1 & 0 & 0 \\ -sinRy & 0 & \cos Ry & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} \times \underset{matOffsetRx}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos Rx & -sinRx & 0 \\ 0 & sinRx & \cos Rx & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} \times$$

$$\underset{matDofRz}{\begin{bmatrix} \cos Rz & -sinRz & 0 & 0 \\ sinRz & \cos Rz & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} \times \underset{matDofRy}{\begin{bmatrix} \cos Ry & 0 & sinRy & 0 \\ 0 & 1 & 0 & 0 \\ -sinRy & 0 & \cos Ry & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} \times \underset{matDofRx}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos Rx & -sinRx & 0 \\ 0 & sinRx & \cos Rx & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} \times \underset{matDofTx}{\begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}$$

La secuencia de operaciones a realizar para la generación y aplicación de la transformación de un nodo *Skeleton* sería el mostrado en l siguiente bloque de pseudocódigo:

```
matOffsetPos = generateTranslateMatrix( offsetPos[X], offsetPos[Y], offsetPos[Z] );
matOffsetRz  = generateRotzMatrix( offsetHpr[0] );
matOffsetRy  = generateRotyMatrix( offsetHpr[1] );
matOffsetRx  = generateRotxMatrix( offsetHpr[2] );
offsetMat    = matOffsetPos × matOffsetRz × matOffsetRy × matOffsetRx;

matDofRz = generateRotzMatrix( dofRz );
matDofRy = generateRotyMatrix( dofRy );
matDofRx = generateRotxMatrix( dofRx );
matDofTx = generateTranslateMatrix( dofTx, 0, 0);
dofsMat  = matDofRz × matDofRy × matDofRz × matDofTx;

sklMat    = offsetMat × dofsMat;
sklAbsMat = getCurrentModelMatrix() × sklMat;.
setCurrentModelMatrix( sklAbsMat);
```

Generación directa de las matrices de transformación de un nodo *Skeleton*.

Como se puede observar, para la generación de la matriz a aplicar por el nodo *Skeleton* (*sklMat*), es necesaria la generación de 8 matrices de transformación y su concatenación (7 multiplicaciones de matrices). Además, es necesaria una multiplicación de matrices adicional, encargada de obtener la matriz absoluta del nodo *Skeleton,* la cual se obtiene multiplicando la matriz del nodo *Skeleton* por la matriz actual existente en ese punto del recorrido del grafo de escena -ésta es la matriz existente en el parte superior de la pila de matrices descrita en el apartado 4.2.3, dicha matriz es obtenida a partir de la función *getCurrentModelMatrix()*-, también es necesario hacer actualizar el contenido de dicha matriz con el resultado de dicha multiplicación -mediante la operación *setCurrentModelMatrix()*.

El procesado de un punto de articulación es pues un proceso muy costoso, en el que puede ser necesaria la realización de 8 multiplicaciones de matrices (512 multiplicaciones y 384 sumas), y además, la generación de las matrices individuales. El nodo *Skeleton* emplea diferentes estrategias para conseguir que el coste computacional asociado a la generación de la matriz de transformación de un nodo *Skeleton* se reduzca considerablemente.

4.3.6 Estrategias para la reducción del coste computacional.

Como se ha descrito en el apartado anterior, la gestión en tiempo real de un nodo *Skeleton* es un proceso muy costoso en el que puede ser necesaria la construcción de 8 matrices de transformación y sus multiplicaciones. La obtención de las matrices de transformación de los nodos *Skeleton* es uno de los aspectos más costosos del procesado en tiempo real de los actores virtuales, y su optimización es uno de los factores que más afectan en la reducción de su coste computacional global. En siguientes apartados se presentan distintas optimizaciones que serán aplicadas de forma sucesiva. En un apartado final se mostrará la mejora computacional introducida por cada uno de ellas.

4.3.6.1 Precálculo de la Matriz de Offset.

Una vez finalizada la fase de definición un nodo *Skeleton*, los valores de los offsets de traslación y rotación permanecen fijos. Por ello es posible calcular dichas matrices de transformación y acumularlas en una única matriz (***offsetMat)*** que es almacenada dentro del nodo *Skeleton*.

Si se dispone de la matriz *offsetMat* que ha sido calculada en una fase de preproceso, la secuencia de operaciones para generar y aplicar las transformaciones de un nodo *Skeleton* sería la siguiente:

```
matDofRz  = generateRotzMatrix( dofRz);
matDofRy  = generateRotyMatrix( dofRy);
matDofRx  = generateRotxMatrix( dofTx);
matDofTx  = generateTranslateMatrix(  dofTx, 0, 0);
dofsMat     = matDofRz x matDofRy x matDofRz x matDofTx;

sklMat      = offsetMat  x  dofsMat;

sklAbsMat = getCurretModelMatrix() x  sklMat;
setCurretModelMatrix( sklAbsMat );
```

Optimización A. Precálculo de la matriz de offsets.

Así pues, en tiempo de ejecución bastaría con generar 4 matrices de transformación y realizar 5 multiplicaciones de matrices (en lugar de las 8 multiplicaciones que eran necesarias en el caso inicial).

4.3.6.2 Cálculo optimizado de la Matriz de Grados de Libertad.

Un nodo *Skeleton* tiene un total de cuatro grados de libertad que serán almacenados en única matriz (*dofsMat*). La forma más sencilla de hacer esto sería generar cada matriz individualmente, y multiplicarlas a continuación para obtener la matriz final. Las operaciones a realizar se representarían de forma matricial de la siguiente manera:

Heading (eje Z) **Pitch (eje Y)** **Roll (eje X)** **Trans (eje X)**

$$\begin{bmatrix} \cos Rz & -sinRz & 0 & 0 \\ sinRz & \cos Rz & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} \cos Ry & 0 & sinRy & 0 \\ 0 & 1 & 0 & 0 \\ -sinRy & 0 & \cos Ry & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos Rx & -sinRx & 0 \\ 0 & sinRx & \cos Rx & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} & Tx \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{bmatrix}$$

Realizar directamente esta secuencia de operaciones es computacionalmente muy costoso. Sin embargo, se puede observar que cualquiera de estas matrices de transformación contiene al menos 10 elementos que tienen un valor 0, y al menos 2 elementos con valor 1. Si se operase directamente con las matrices sin tener en cuenta esta característica, alrededor de un 75% del tiempo consumido en cada multiplicación entre matrices sería innecesario.

Si las cuatro matrices son concatenadas de forma algebraica, se obtiene la siguiente expresión:

Matriz de heading-pitch-roll-tx

$$\begin{bmatrix} \cos Rz * \cos Ry & \cos Rz * sinRy * sinRx - sinRz * \cos Rx & \cos Rz * sinRy * \cos Rx + sinRz * sinRx & \cos Rz * \cos Ry * Tx \\ sinRz * \cos Ry & sinRz * sinRy * sinRx + \cos Rz * \cos Rx & sinRz * sinRy * \cos Rx - \cos Rz * sinRx & sinRz * \cos Ry * Tx \\ -sinRy & \cos Ry * sinRx & \cos Ry * \cos Rx & -sinRy * Tx \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Esta matriz aún puede ser representada de una forma más sencilla haciendo que los cálculos trigonométricos se realize en una fase previa a la generación de la matriz, el resultado de esta matriz simplificada de *heading-pitch-roll-tx* sería el siguiente:

Matriz simplificada de heading-pitch-roll-tx

$A = cosRz;\ B = cosRy;\ C = cosRx$

$D = sinRz;\ E = sinRy\ ;\ F = sinRx$

$$\begin{bmatrix} A*B & A*E*F - D*C & A*E*C + D*F & A*B*Tx \\ D*B & D*E*F + A*C & D*E*C - A*F & D*B*Tx \\ -E & B*F & B*C & -E*Tx \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Como se puede observar, para la generación de esta matriz tan sólo serían necesarias 6 operaciones de calculo de senos y cosenos, 21 multiplicaciones y 4 sumas. Si esta matriz hubiese sido generada a través de multiplicaciones estándar entre 4 matrices de 4x4 hubiese sido necesario realizar 192 multiplicaciones y 144

sumas. La utilización directa de esta nueva matriz *heading-pitch-roll-tx* hace que el coste derivado de la generación de se vea reducido en un factor de 10.

Hasta ahora se ha estado considerando que el nodo *Skeleton* emplea sus 4 grados de libertad, sin embargo, ésta es una circunstancia en la práctica poco frecuente La mayoría de nodos *Skeleton* no tiene 4 grados de libertad, sino 2 o incluso 1. El hecho de que alguno de los grados de libertad no estén presentes genera versiones más sencillas de la matriz anterior, haciendo que el coste computacional pueda ser reducido de nuevo. La tabla que sigue muestra distintas posibles combinaciones de grados de libertad, así como los métodos para generar sus matrices y su coste computacional correspondiente.

heading-pitch-roll-tx	$AE = A*E;\ DE = D*E;\ AB = A*B;\ DB = D*B$ $\begin{bmatrix} AB & AE*F - D*C & AE*C + D*F & AB*Tx \\ DB & DE*F + A*C & DE*C - A*F & DB*Tx \\ -E & B*F & B*C & -E*Tx \\ 0 & 0 & 0 & 1 \end{bmatrix}$	6 ops. sin/cos. 17 multiplicaciones. 4 sumas.
heading-pitch-roll $Tx = 0$	$AE = A*E;\ DE = D*E$ $\begin{bmatrix} A*B & AE*F - D*C & AE*C + D*F & 0 \\ D*B & DE*F + A*C & DE*C - A*F & 0 \\ -E & B*F & B*C & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	6 ops. sin/cos 14 multiplicaciones 4 sumas.
heading-roll-tz $B=1\ ;\ E=0$	$\begin{bmatrix} A & -D*C & D*F & A*Tx \\ D & A*C & -A*F & D*Tx \\ 0 & F & C & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	4 ops. sin/cos. 7 multiplicaciones.
heading-roll $B=1\ ;\ E=0;\ Tx=0$	$\begin{bmatrix} A & -D*C & D*F & 0 \\ D & A*C & -A*F & 0 \\ 0 & F & C & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	4 ops. sin/cos. 5 multiplicaciones.
heading-tx $B=1\ ;\ E=0$ $C=1;\ F=0$	$\begin{bmatrix} A & -D & 0 & A*Tx \\ D & A & 0 & D*Tx \\ 0 & F & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	2 ops. sin/cos. 2 multiplicaciones.
heading $B=1\ ;\ E=0$ $C=1;\ F=0;\ Tz=0$	$\begin{bmatrix} A & -D & 0 & 0 \\ D & A & 0 & 0 \\ 0 & F & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	2 ops. sin/cos. 0 multiplicaciones.

Tabla 4-1.Distintas especializaciones de la matriz de Dofs de un nodo *Skeleton*

Las combinaciones no mostradas en la tabla anterior (*heading-pitch-roll-tx*, *heading-pitch-roll*, *heading-pitch-tx*, etc) son obtenidas de un modo equivalente. El campo ***dofsCfg*** del nodo *Skeleton* es una máscara de bits que almacena la configuración de grados de libertad empleados y sirve pare seleccionar de un modo rápido el método empleado para la generación de la matriz de dofs.

Si se considera que la generación de la matriz de dofs mediante este método es llevada a cabo por una función de nombre *computeSklDofsMatrix()*, la secuencia de operaciones a realizar en un nodo *Skeleton* pasaría a ser la siguiente:

```
dofsMat      = computeSkldofsMatrix( dofRz, dofRy, dofRx, dofTx, dofsCfg);
sklMat       = offsetMat  x  dofsMat.

sklAbsMat  =getCurrentModelMatrix()  x  sklMat;
setCurrentModelMatrix( sklAbsMat);
```

Optimización B. Empleo de matriz simplificada de heading-pitch-roll-tx.

4.3.6.3 Bloqueo en los cálculos por coherencia temporal.

Según lo visto hasta ahora, el procesado de cada nodo *Skeleton* lleva implícita la generación de la *matriz de dofs*, y, además, un par de multiplicaciones de matrices: La multiplicación de la *matriz de offsets* por la *matriz de dofs* y la multiplicación del resultado de ésta última por la *matriz actual.*

En principio estamos suponiendo que en cada frame se modifican los valores de todos los nodos *Skeleton*, pero esto ocurre raras veces en la practica. Por ejemplo, una persona que escribiendo a máquina modifica en cada instante las posiciones de sus dedos y sus ojos, pero sus piernas y columna vertebral permanece inmóviles, en estas articulaciones no sería necesario recalcular sus matrices puesto que son válidas las calculadas en frames anteriores. Para poder tener en cuenta esta situación se ha introducido en los nodos *Skeleton* el campo ***dirty***, que indica si los valores de la articulación han de ser recalculados o no. En el momento en el que es fijado el valor de algún grado de libertad, se comprueba si es distinto del valor previo, y en tal caso este flag es puesto a un valor distinto de 0. Cuando un nodo *Skeleton* va a ser aplicado, se comprueba si su flag de *dirty* es distinto de 0, en tal caso, la matriz del nodo *Skeleton* es recalculada y almacenada en una matriz total (*sklMat*), y el flag *dirty* es puesto de nuevo a 0.

El siguiente fragmento de código, (*Optimización C*) indica esquemáticamente la secuencia ejecutada por cada nodo *Skeleton* en cada frame:

```
if ( dirty ) {
        dofsMat = computeSkldofsMatrix( dofRz, dofRy, dofRx, dofTx, dofsCfg);
        sklMat    = offsetMat x dofsMat;
        dirty = FALSE;
        }

sklAbsMat  =getCurrentModelMatrix()  x  sklMat;
setCurrentModelMatrix( sklAbsMat);
```

Optimización C. Bloqueo de cálculos por coherencia temporal.

La utilización de esta estrategia hace que los costes derivados del procesado de los nodos *Skeleton,* se reduzcan considerablemente en el caso de actores que alternan periodos de movimiento, con periodos de reposo, o de que los actores presenten partes de su cuerpo que permanezcan temporalmente inmóviles.

4.3.6.4 Empleo de funciones específicas para la multiplicación de matrices.

En la secuencia de operaciones mostrada en la *Optimización C*, la mayoría del tiempo se está consumiendo en la realización de la multiplicación de la *matriz de offset* por la *matriz de dofs*, y en la multiplicación de la matriz *sklMat* por la *matriz actual.*

La obtención de la matriz *sklMat* a partir de la multiplicación directa de las dos matrices tendría un coste elevado (64 multiplicaciones y 48 sumas), afortunadamente, tanto la *matriz de offset* como las *matrices de dofs* presentan peculiaridades que permiten que sea posible realizar estas multiplicaciones con coste computacional mucho menor: en la *matriz de offsets* (*offsetMat*) la última columna está formada por tres ceros y un uno. La solución consiste en desarrollar una desarrollar una función -de nombre *composeSklMatrix()*- que realice la multiplicación de la *matriz offset* por la *matriz de dofs* teniendo en cuenta estas características. Así la multiplicación expresada como:

$$\mathbf{sklMat} = \mathbf{offsetMat} \times \mathbf{dofsMat}$$

Pasaría a ser expresada como:

$$\mathbf{sklMat} = \mathbf{composeSklMatrix(offsetMat,\ dofsMat,\ dofsCfg)}$$

En dicha función se tendrá en cuenta el número y tipo de los grados de libertad del nodo *Skeleton* (*dofsCfg*) para realizar las multiplicaciones con un coste mínimo. En la siguiente tabla se muestra el coste computacional de realización de algunas de estas multiplicaciones.

heading-pitch-roll-tx	36 multiplicaciones y 27 sumas.
heading-pitch-roll	36 multiplicaciones y 27 sumas.
heading-roll-tx	32 multiplicaciones y 23 sumas.
heading-roll	24 multiplicaciones y 15 sumas.
heading-tx	18 multiplicaciones y 12 sumas.
heading	18 multiplicaciones y 12 sumas.

Tabla 4-2. Costes de distintas especializaciones de la función *composeSklMatrix()*.

De un modo similar, se puede reducir el coste de la generación de la matriz absoluta del nodo *Skeleton* (*sklAbsMat*) teniendo en cuenta que tanto la matriz actual como la matriz *sklMat* son matrices afines en las que su última fila está formada por 3 ceros y un 1, así la expresión:

$$\mathbf{currentMat} = \mathbf{currentMat} \times \mathbf{sklMat}$$

Será sustituida por:

$$\mathbf{currentMat} = \mathbf{composeAffineMatrix(\ currentMat\ ,\ sklMat)}$$

El coste de realización de esta multiplicación de matrices afines es de 36 multiplicaciones y 27 sumas, frente a las 64 multiplicaciones y 48 sumas de una multiplicación genérica de matrices.

Con estas dos modificaciones (*Optimización D*), la secuencia de operaciones a realizar durante el procesado de uno nodo *Skeleton* pasaría a ser:

```
if ( dirty ) {
        dofsMat = computeSkldofsMatrix( dofRz, dofRy, dofRx, dofTx, dofsCfg);
        sklMat   =  composeSklMatrix( offsetMat, dofsMat, dofsCfg);
         dirty =FALSE;
        }

sklAbsMat  = composeAffineMatrix ( getCurrentModelMatrix() ,  sklMat);
setCurrentModelMatrix( sklAbsMat);
```

Optimización D. Empleo de funciones especificas de multiplicación de matrices.

4.3.6.5 Empleo selectivo del hardware de multiplicación de matrices.

Como se está viendo, el cuello de botella del procesado de un nodo *Skeleton* está en la multiplicación de matrices. Es posible hacer que el proceso de aplicación de transformaciones por parte del nodo *Skeleton* presente mejores resultados recurriendo a la utilización de hardware gráfico con capacidad para realizar multiplicaciones de matrices. Esta característica hace que ciertas operaciones de OpenGl relacionadas con la gestión de las matrices de transformación tales como *glRotate()*, *glTranslate()*, *glMultmatrix()*, *glPushMatrix()* o *glPopMatrix(),* sean realizadas directamente en el hardware de la tarjeta, liberando a la CPU de este trabajo y acelerando el proceso total.

Para emplear de forma adecuada el hardware de multiplicación de matrices, sin perder la capacidad de aprovechar la coherencia temporal habría que transformar el código mostrado en *Optimización D* en lo siguiente:

```
switch (dirty){
    case 2: dofsMat = computeSkldofsMatrix(dofRz, dofRy, dofRx, dofTx, dofsCfg );
              glMultMatrix( offsetMat );
              glMultMatrix( dofsMat );
              dirty = 1;
              break;
   case 1:  sklMat = composeSklMatrix( offsetMat, dofsMat, dofsCfg);
              glMultMatrix( sklMat );
              dirty = 0;
              break;
   case 0: glMultMatrix( sklMat );
              break;
}
```

Empleo de la capacidad de multiplicación de matrices en el hardware gráfico.

Como se puede observar el flag *dirty* presenta un comportamiento un poco más complejo: cuando un nodo *Skeleton* es modificado, su valor es fijado a 2, y permanece en este estado mientras que se estén realizando actualizaciones sobre él. Tan pronto como este *Skeleton* deja de ser modificado, pasa por un estado transitorio (*dirty* = 1), en el cual en el cual se calcula el valor de *sklMat* que será empleado mientras que el nodo *Skeleton* permanezca sin modificación.

Un nodo *Skeleton* estará habitualmente con su flag de *dirty* igual a 2 (está siendo modificado), o a 0 (permanece sin modificación). Como se puede observar, en ambos casos las funciones de multiplicación de matrices en CPU *composeSklMatrix()* y *composeAffineMatrix()* han sido sustituidas por ordenes *glMultMatrix()* que se realizan en el hardware gráfico de un modo más eficiente.

La orden *glMultMatrix()* multiplica una matriz por la matriz existente en la pila interna de OpenGl, esto hace que no sea necesario llamar a la operación *setCurrentModelMatrix()* -que actualizaba la pila del *grafo de escena* y también la de OpenGl-, y también evita que en el caso de *dirty* tenga un valor de 2 no sea necesario generar una matriz intermedia *sklMat*.

El principal inconveniente de la realización de las multiplicaciones de matrices en el hardware gráfico es que el resultado de dicha operación no es conocido por la CPU - el problema era la lentitud de la orden *glGet(GL_MODELVIEW_MATRIX)* -, lo cual es necesario en el caso de que se quiera conocer la matriz absoluta del nodo *Skeleton* (la matriz que indica su posición con respecto al sistema de coordenadas del mundo). Si bien esta circunstancia no es habitual, existen ciertas operaciones en las que es totalmente necesario (detección de colisiones, cálculos de cinemática inversa...). Los nodos *Skeleton* lleva un flag (***trackSkeleton***) con el cual se marcan los nodos *Skeleton* cuya posición absoluta necesita ser conocida. El hecho de decirle a un nodo *Skeleton* que es necesario conocer su matriz absoluta, fuerza a que todos los nodos *Skeleton* de los que depende jerárquicamente tengan que fijar también este modo.

La siguiente modificación sobre el fragmento de código anterior permite compatibilizar la necesidad de conocimiento de las matrices absolutas con el aprovechamiento del hardware de multiplicación de matrices(*Optimización E*):

```
if  (trackSkeleton){
        if ( dirty == 2  ){
            dofsMat = computeSkldofsMatrix( dofRz, dofRy, dofRx, dofTx, dofsCfg);
            sklMat   = composeSklMatrix( offsetMat, dofsMat, dofsCfg);
            dirty = 0;
        }
        sklAbsMat  = composeAffineMatrix (getCurrentModelMatrix() ,  sklMat);
        setCurrentModelMatrix( sklAbsMat );
}else{
        switch (dirty){
            case 2:   dofsMat = computeSkldofsMatrix( dofRz, dofRy, dofRx, dofTx, dofsCfg);
                      glMultMatrix(offsetMat);
                      glMultMatrix(dofsMat);
                      dirty = 1;
                      break;
            case 1:   sklMat = composeSklMatrix( offsetMat, dofsMat, dofsCfg);
                      glMultMatrix(sklMat)
                      dirty = 0;
                      break;
            case 0:   glMultMatrix(sklMat);
                      break;
        }
}
```

Optimización E. Empleo selectivo de hardware de multiplicación de matrices.

4.3.6.6 Estimación de la mejora.

Mediante el empleo de las optimizaciones anteriores, se consigue una importante reducción en el coste de generación y aplicación de la matriz de transformación generada por un nodo *Skeleton*. En este apartado se va a realizar una estimación de la diferencia de coste temporal existente entre la aplicación directa de todas las

operaciones de un nodo *Skeleton* sin ningún tipo de optimización, y aplicando todas las estrategias de reducción de coste (*Optimización E.*). También vamos a ver la mejora computacional introducida por cada optimización.

La mayoría de operaciones implicadas en el procesado de las matrices de un nodo *Skeleton* son operaciones de seno/coseno, multiplicaciones y sumas. Para simplificar la evaluación de los costes vamos a suponer que la realización de cada una de estas operaciones tienen un coste temporal equivalente al que vamos identificar con el término *FMULT* (coste de realización de una multiplicación entre dos números en punto flotante). Se va también a considerar que en una escena típica la mayoría de las articulaciones de los actores tienen dos grados de libertad rotacionales.

Para analizar la mejora del ahorro computacional va ha ser necesario aislar los distintos tipos de operaciones que intervienen en gestión de las transformaciones aplicadas por un nodo *Skeleton*. En la siguiente tabla se muestran dichas operaciones con una abreviatura que las identifica y sus costes parciales:

costeGenRotMat	Coste medio de generación de una matriz de rotación. 2 ops. sin/cos.	2 FMULTs
costeGenDofsMat	Coste medio de generación de la matriz de dofs mediante la función *computeSkldofsMatrix()*. El coste de generación de una matriz de heading-roll puede ser sacado de la *Tabla 4-1* y es de 4 ops. sin/cos y 5 mults .	9 FMULTs
costeMultMat	Coste de multiplicación de dos matrices de 4x4 genéricas en CPU. 64 mults y 48 sumas	112 MULTs
costeMultHard	Coste de realización de una multiplicación de matrices en hardware mediante la función *glMultMatrix()*[3].	10 FMULTs
costeMultDofs	Coste medio de generación de la matriz de dofs mediante la función *composeSklMatrix()*. El Coste de multiplicación de una matriz de heading-roll por la matriz de offsets puede ser extraído de *Tabla 4-2* (24 mults y 15 sumas).	39 FMULTs
costeMultAfin	Coste de multiplicación de dos matrices afines mediante la operación *composeAffineMatrix()*. 36 mults y 27 sumas.	63 FMULTs

Tabla 4-3. Coste de las distintas operaciones básicas realizadas en el procesado de las matrices de transformación de un nodo *Skeleton con dos grados de libertad rotacionales*.

[3] La multiplicación de dos matrices de 4x4 en el hardware gráfico se realiza mediante procesadores específicos y su coste no puede ser medido en las mismas unidades que las operaciones que se realizan en la CPU, sin embargo si que es posible establecer un coste equivalente basándonos en un pequeño test que realice multiplicaciones de matrices en el hardware gráfico mediante la operación *glMultMatrix().* Este test ha sido realizado en una Onyx2 IR R10000 195Mhz y el resultado es que cada multiplicación tiene un coste equivalente a 10 FMULTs.

Vamos a expresar el coste original del procesado del nodo *Skeleton* en función de estas abreviaturas.

Caso Original.

Generación de 6 matrices de rotación y realización de 8 multiplicaciones de matrices de 4x4

coste = 6 x costeGenRotMat + 8 x costeMultMat

Optimización A.

Precálculo de la *matriz de offsets*. El coste ahora se podría expresar como:

coste = 3 x costeGenRotMat + 5 x costeMultMat

Optimización B.

Cálculo optimizado de la *matriz de DOFs*. Ahora el coste sería el coste medio de generación de la *matriz de dofs*, más el coste realizar dos multiplicaciones de matrices 4x4:

coste = costeGenDofsMat + 2 x costeMultMat

Optimización C.

Utilización de la coherencia temporal.

coste = fCoherencia x (costeGenDofsMat + costeMultMat) +costeMultMat.

Optimización D.

Utilización de funciones de multiplicación de matrices especificas.

coste = fCoherencia x (costeGenDofsMat + costeMultDofs) + costeMultAfin

Optimización E.

Empleo del hardware de multiplicación de matrices. La estimación del coste de la *Optimización E* es más difícil, ha de ser calculado a partir de dos costes parciales, el coste sin tracking, y el coste con tracking, de tal modo que el coste de procesado con la optimización E se podría expresar como:

coste = fTracking x costeConTracking + (1 -fTracking) x costeSinTracking

El coste sin tracking podría expresarse como:

costeSinTracking = fCoherencia x (costeGenDofsMat +costeMultHard) + costeMultHard.

y el coste con tracking como:

costeConTracking = fCoherencia x (costeGenDofsMat + costeMultDofs) + costeMultAfin,

Si en dichas expresiones se emplean los costes medidos en *FMULTs* de la *Tabla 4-3* se obtiene:

costeSinTracking = fCoherencia x 19 + 10.

costeConTracking = fCoherencia x 48 + 63

A partir de lo cual se puede obtener la siguiente expresión del coste total:

coste = fTracking x (fCoherencia x 48 + 63) + (1 - fTracking) x (fCoherencia x 19 + 10)

Lo cual también puede ser expresado como:

coste = fTracking x fCoherencia x 29 + fTracking x 53 + fCoherencia x 19 +10.

Esta expresión define el coste final de procesado de las transformaciones de un nodo *Skeleton* una vez aplicadas todas las optimizaciones, y depende del *factor de coherencia temporal* y también del *factor de tracking*. La representación gráfica de dicha expresión sería:

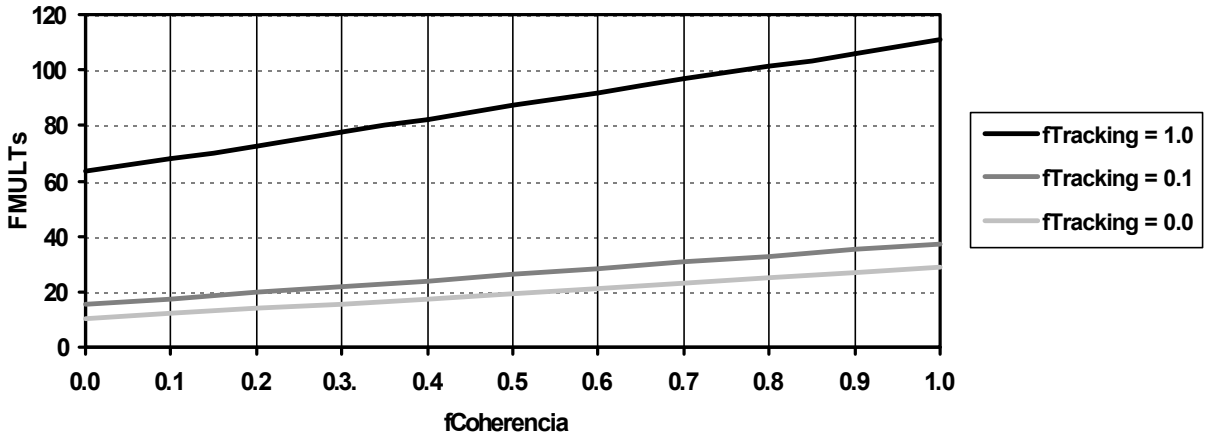


Figura 4-16. Coste de procesado de las transformaciones de un nodo *Skeleton* en función de los *factores de Coherencia* y *Tracking*.

Como se puede observar en la gráfica anterior. En el caso de que el hardware gráfico con aceleración en la multiplicación de matrices, se puede conseguir que el procesamiento de las transformaciones de un nodo *Skeleton* sea hasta 4 veces más rápido (caso de *fTracking* = 0). El caso de *fTracking* = 1 se daría en el caso de que las multiplicaciones de matrices no pudiesen ser realizadas en el Hardware gráfico (bien porque no disponga de esta capacidad, o bien porque por alguna razón sea necesario conocer la posición absoluta de todos los puntos de articulación).

Si calculamos coste del procesado de la transformaciones de un nodo *Skeleton* teniendo en cuenta las distintas optimizaciones, y suponiendo una escena típica con de *factores de Coherencia* entre 0.7 y 0.2, y *factores de Tracking* entre 0.1 y 0, obtenemos la siguiente tabla:

	coste
caso Original	908 FMULTs
Optimización A	566 FMULTs
Optimización B	233 FMULTs
Optimización C. (fCoherencia = 0.7)	193 FMULTs
Optimización C (fCoherencia = 0.2)	136 FMULTs
Optimización D (fCoherencia = 0.7)	97 FMULTs
Optimización D (fCoherencia = 0.2)	73 FMULTs
Optimización E (fCoherencia = 0.7, fTracking = 0.1)	31 FMULTs
Optimización E (fCoherencia = 0.7, fTracking = 0.0)	23 FMULTs
Optimización E (fCoherencia = 0.2, fTracking = 0.1)	20 FMULTs
Optimización E (fCoherencia = 0.2, fTracking = 0.0)	14 FMULTs

Tabla 4-4. Comparativa de coste de procesado de las transformaciones de un nodo *Skeleton* con las distintas optimizaciones.

En la tabla anterior se muestra como en una escena típica se puede conseguir que el procesado de las transformaciones se realice entre **9 y 12** veces más rápido en el caso de no disponer de hardware de multiplicación de matrices (sólo se puede llegar hasta la *Optimización D*) y entre **30 y 60** veces más rápido en el caso de disponer de hardware de multiplicación de matrices.

4.4 Nodo *Actor*.

La estructura articulada de un actor virtual puede ser totalmente descrita mediante la utilización de nodos *Skeleton*, sin embargo, tal y como ya se ha introducido en el apartado 4.2.2, además es necesario disponer de un nodo adicional que permita hacer que un actor virtual sea algo más que un conjunto de nodos *Skeleton* dispersos por el *grafo de escena.* El nodo *Actor*, además de agrupar de manera lógica a todos los nodos que pertenecen a un determinado actor virtual, tiene las siguientes misiones:

- Actúa como sistema de referencia base para el resto de nodos que componen al actor virtual.
- Permite definir la posición del actor virtual dentro de la escena.
- Actúa como centro de control sobre el actor virtual, dirigiendo el funcionamiento del resto de nodos que componen un actor virtual, y sirviendo como estructura de soporte para la aplicación de métodos especiales de gestión de gestión de niveles de detalle y culling, y para la gestión de su comportamiento.

En los siguientes apartados se describe la forma en la que el nodo *Actor* actúa como sistema de referencia base para los nodos del actor virtual, su utilización para definir la ubicación del actor virtual en la escena, y también su función como centro de control sobre el resto de nodos que lo componen, sus métodos de gestión de culling y LOD, y su gestión del comportamiento. A continuación se muestra su estructura de datos, se presenta el problema computacional existente a nivel del procesado de sus transformaciones, y se muestran soluciones. En un último apartado se presenta de una forma integrada cual es la secuencia de operaciones que es realizada por un nodo *Actor*.

4.4.1 Nodo Actor como sistema de referencia base para el resto de nodos de un actor virtual.

El nodo *Actor* actúa siempre como raíz para el resto de nodos del actor virtual, y contiene un punto tridimensional, de nombre ***Skeletons Root***, que actúa como sistema de referencia base para la definición del resto de la estructura articulada del actor virtual**.** En la *Figura 4-17* se puede observar la organización jerárquica de un actor estructurada en torno al *Skeletons Root* del nodo Actor, y a varios nodos *Skeleton*.

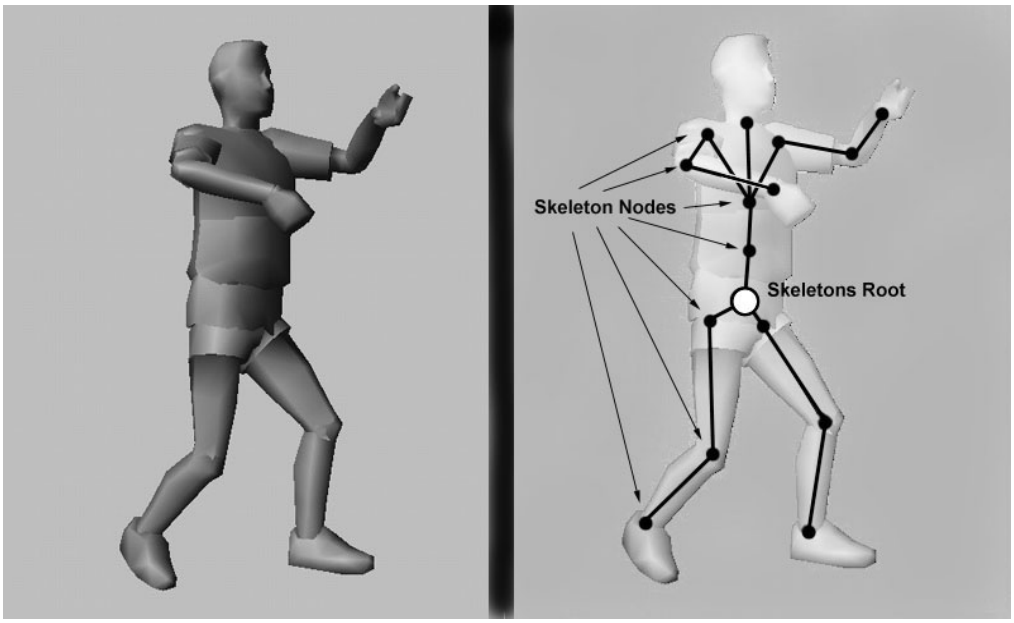


Figura 4-17. Estructura articulada de un actor virtual definida a partir de un grupo de nodos *Skeleton* y un *Skeletons Root.*

4.4.2 Nodo Actor como elemento ubicador del actor virtual en la escena.

El segundo objetivo de un nodo *Actor* es definir la posición del actor virtual dentro del espacio tridimensional, esto puede ser conseguido directamente por medio de la modificación de la posición espacial del *Skeletons Root* tal y como se muestra en la *Figura 4-18*.

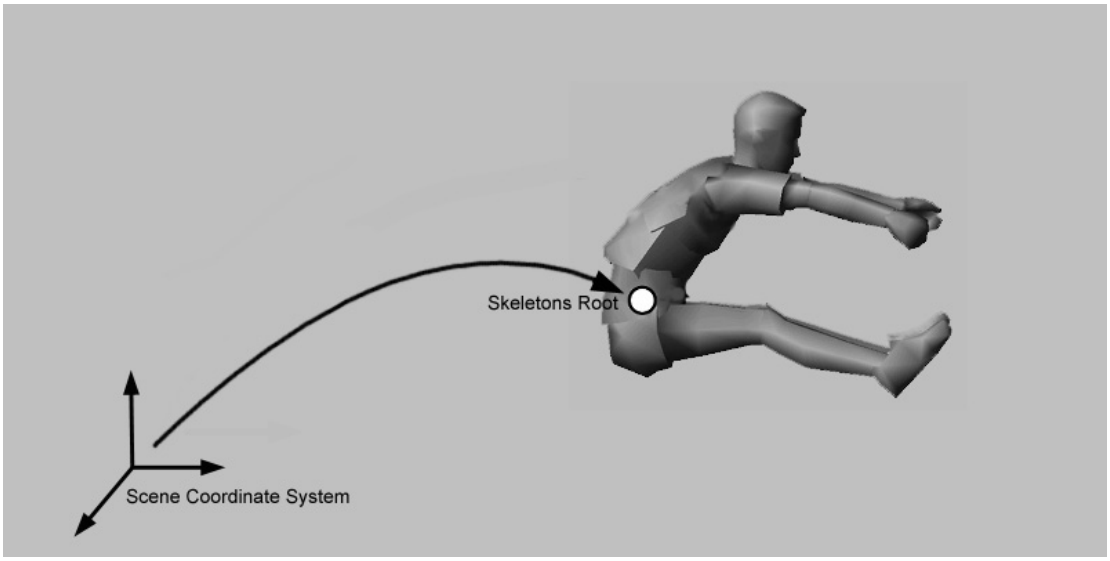


Figura 4-18. El *Skeletons Root* almacenango la información de la ubicación del actor dentro de la escena.

Si se siguiese esta técnica, la posición de un actor en el espacio sería definida a partir de 6 parámetros, 3 de ellos definen la traslación y otros 3 serían los *ángulos de Euler* que definen la orientación.

Este tipo de caracterización de puede resultar adecuada para un programa de animación 3D tradicional, sin embargo, resulta insuficiente para representar de forma adecuada la posición de un actor sintético TR. El *Skeletons Roots* es necesario como sistema de referencia para el resto de nodos *Skeleton*, pero no resulta cómodo a la hora de ubicar al actor en una posición concreta de la escena. En este sentido, resultaría mucho más útil poder especificar directamente la posición de su centro de masas o de un punto de contacto con el suelo. Esto se puede conseguir haciendo que el *Skeletons Root* dependa de otro sistema de referencia auxiliar al que llamaremos ***Reference Point***.

En la *Figura 4-19* se pueden observar cuatro ejemplos distintos de utilización del *Reference Point*.

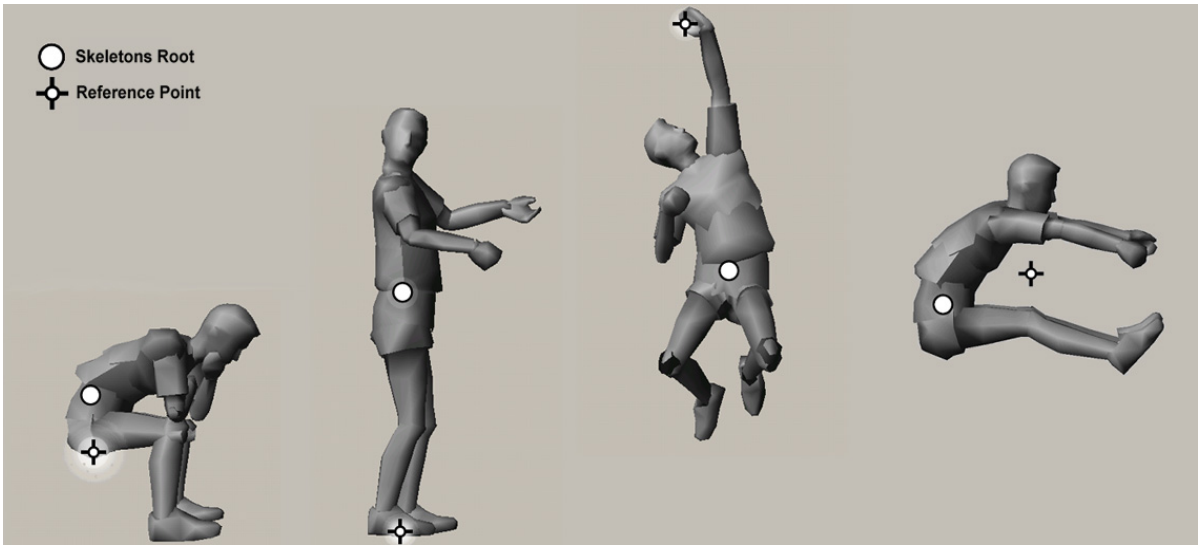


Figura 4-19. Ejemplos de ubicación del Reference Point en distintas posiciones respecto al actor.

En el primer actor de la imagen se ha utilizado el *Reference Point* para colocar al actor sentado sobre una posición espacial concreta. En el segundo, el *Reference Point* define el punto de contacto de los pies del actor con el suelo, lo que permite ubicar rápidamente al actor en un punto sobre el suelo de la escena 3D. El tercer ejemplo muestra como se puede facilitar la colocación del actor de modo que apareciese agarrado a una rama, se podría definir fácilmente el balanceo del actor en la rama actuando sobre la posición de *Skeletons Root* respecto al *Reference Point*. Por último se muestra un ejemplo en el que el *Reference Point* es utilizado para representar el centro de masas del a*ctor*, este ejemplo puede representar un instante del movimiento de un saltador de longitud, en este caso, el *Reference Point* seguiría una trayectoria parabólica, y la posición del *Skeletons Root* respecto al *Reference Point* se modificaría dependiendo de la disposición corporal del actor. Este último caso es mostrado con más detalle en la *Figura 4-200*, en ella se puede observar como *Reference Point* define la ubicación del centro de masas del actor, y también como se aplican dos transformaciones, la primera de ellas define la posición del centro de masas del actor en la escena (en este caso siguiendo una trayectoria parabólica), y la segunda define la posición del *Skeletons Root* respecto a dicho centro de masas.

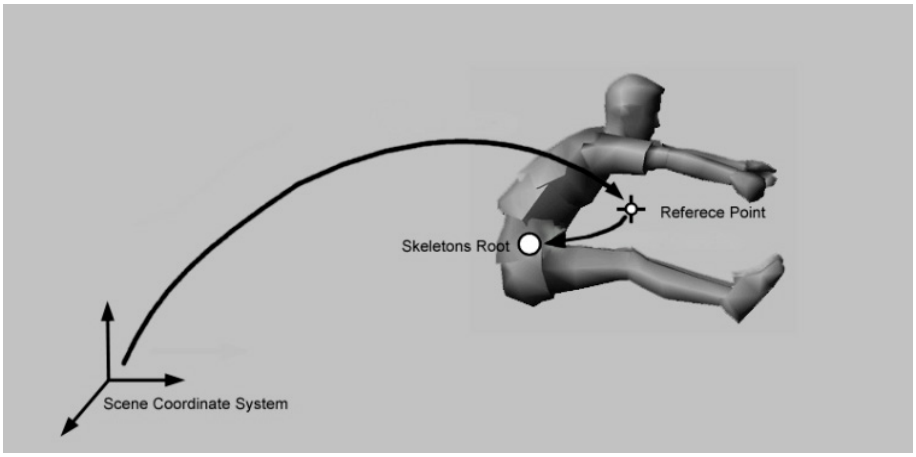


Figura 4-20. Ubicación del actor en la escena empleando un doble sistema de transformaciones.

La existencia de un doble sistema de coordenadas presenta las siguientes ventajas:

- Permite definir la posición del actor en la escena de una forma intuitiva.
- Permite almacenar secuencias de movimiento que pueden ser repetidas en cualquier punto de la escena, por ejemplo un actor humanoide que ejecute la acción de sentarse en una silla (se almacenaría como modificaciones del *Skeletons Root,* y valor del *Reference Point* definiría el punto en que se quiere que se ejecuten).
- Permite mantener un control independiente de los movimientos del actor, así por ejemplo, en el caso de un actor humano caminando por una escena, el movimiento del *Skeletons Root* definiría la secuencia cíclica ejecutada por de su cadera, mientras que la posición del *Reference Point* podría ser controlada por el desarrollador de la aplicación para definir su movimiento a lo largo de la escena.

Las transformaciones que definen la posición del *Skeletons Root* y del *Reference Point* son especificadas de una forma similar al offset de un nodo *Skeleton,* es decir, una traslación seguida por tres rotaciones codificadas según los *Angulos de Euler*. La posición y orientación del *Reference Point* de un nodo *Actor* es almacenada como dos vectores de tres *floats* en los campos en los campos ***refpointPos*** y ***refpointHpr*** del nodo *Actor*. Esta nomenclatura es traducida internamente a una matriz y almacenada en el campo ***refpointMat***. Algo similar ocurre con el *Skeletons Root*, cuya ubicación en el espacio es definida mediante los campos ***sklsrootPos***, ***sklsrootHpr*** y ***sklsrootMat***.

4.4.3 Nodo Actor como Centro de Control.

El nodo *Actor*, además de servir para definir la posición de un actor virtual en la escena y de servir como sistema de referencia base para el resto de nodos que componen, actúa como centro de control de todo el *grafo de escena* que de él depende, esto permite que el desarrollador de una aplicación pueda controlar totalmente a un actor virtual accediendo de forma única su nodo *Actor.* Además, el nodo *Actor* actúa como centro de gestión de métodos específicos de culling y nivel de detalle, y también como elemento gestor de su comportamiento. Los siguientes subapartados muestran estas características.

4.4.3.1 Nodo *Actor* como elemento de acceso a los nodos *Skeleton*.

La posición de un actor virtual en la escena viene determinada por los valores de las matrices *refpointMat* y *sklsrootMat* del nodo *Actor,* y su postura viene definida por los valores de los grados de libertad aplicados por los nodos *Skeleton*.

Las operaciones de bajo nivel que un usuario puede necesitar en referencia a un nodo *Skeleton* pueden ser de dos tipos:

- Modificar o consultar el valor de alguno de sus grados de libertad.
- Preguntar por la posición espacial de una determinada articulación.

Resulta muy adecuado que el usuario pueda realizar cualquiera de estas dos operaciones accediendo tan sólo al nodo *Actor*, sin necesidad de tener que realizar ningún tipo de operación con los nodos *Skeleton*. El nodo *Actor* actúa como interfaz de manejo de su estructura jerárquica de tal modo que centraliza las acciones o consultas de un usuario necesite realizar sobre un actor virtual. El nodo *Actor* tiene dos vectores encargados proporcionar esta capacidad, el primero de ellos (de nombre ***conf***) es un array de elementos de tipo *float* que almacena los valores de los grados de libertad del actor. El segundo de ellos (de nombre ***lSkels***) es un array de punteros a sus nodos *Skeleton*.

En cada celda del array *conf* se almacena el valor de un grado de libertad, los campos de los nodos *Skeleton* que almacenan el valor de los grados de libertad (*dofRz, dofRy*, *dofRx* y *dofTx)* son en realidad punteros a determinadas celdas de este array.

.

En la *Figura 4-211*, se representa un actor muy sencillo formado por tan sólo cinco nodos *Skeleton*, y que emplea trece grados de libertad. Se puede observar como el campo *conf* del nodo *Actor* es un array de tamaño trece, el cual contiene los valores de todos los grados de libertad. Los nodos *Skeleton* acceden dichos valores por medio de sus correspondientes punteros.

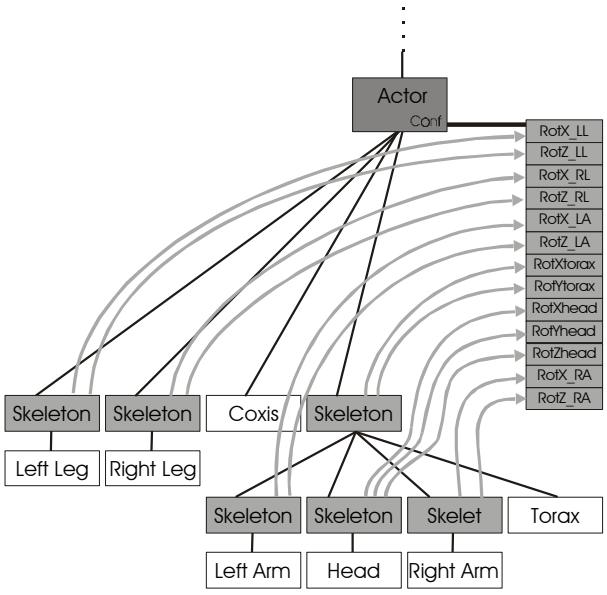


Figura 4-21. Almacenamiento de los grados de libertad de los nodos *Skeleton* en el nodo *Actor*.

Tal y como se muestra en la figura anterior, para conocer o modificar el valor de un determinado *grado de libertad* de un actor virtual no es necesario acceder a los nodos *Skeleton*, tan sólo es necesario acceder a los valores almacenado en la tabla *conf* del nodo *Actor*.

Si bien la mayoría de operaciones con los actores virtuales, acaban reducidas a simples modificaciones sobre los valores de sus grados de libertad, en algunos casos también puede ser necesario acceder a los nodos *Skeleton* para realizar otro tipo de operaciones, (por ejemplo conocer la posición espacial en la que se encuentran, para determinar si se ha producido una colisión con un objeto). El campo ***lSkels*** del nodo *Actor* es array con punteros a todos los nodos *Skeleton* que componen un actor virtual. Mediante este array es posible realizar accesos rápidos a los nodos *Skeleton* sin necesidad de tener que conocer su organización jerárquica. En la *Figura 4-22* se puede observar una descripción total de las relaciones existentes entro los nodos A*ctor* y S*keleton*.

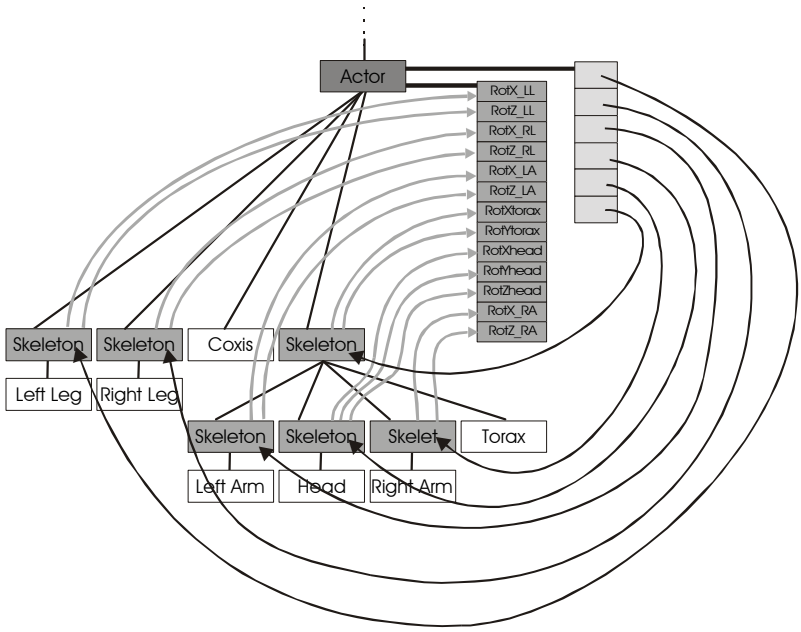


Figura 4-22 . Diagrama total de relaciones existentes entre nodos *Actor* y *Skeleton*.

La existencia de estos dos arrays, además, facilita la creación de una interfaz de gestión de los actores virtuales en el cual tanto los grados de libertad como los puntos de articulación pueden ser identificados mediante índices.

4.4.3.2 Nodo *Actor* como Gestor de métodos de Culling y LOD.

La aplicación de los métodos de culling y LOD tradicionales sobre un actor virtual no resultan adecuados, por esta razón en este trabajo se proponen métodos específicos. Si bien éste es un aspecto que será tratado en detalle en un apartado posterior, vamos a adelantar que el nodo *Actor* será responsable de realizar parte de esta gestión, y almacenara varios campos dedicados a este fin.

Respecto a la gestión de niveles de detalle, el nodo Actor almacenara los campos ***eyeDistance*** y ***priority***, el primero almacena la distancia del *Skeletons Root* del actor al punto de vista, y es empleado en los cálculos encargados de seleccionar los niveles de detalle adecuados. El campo *prioriy* es un valor normalizado que

modula el nivel de detalle empleado para representar a un actor un actor en la escena, de este modo se puede actuar sobre la calidad de representación de los actores que presenten una importancia especial.

El nivel de detalle con el que se represente un actor virtual será controlado a partir de los valores de *distancia* a la cámara y *prioridad* almacenados en el nodo *Actor*. La geometría que representa a un actor cambiara de forma escalonada con la distancia, y el nodo *Actor* almacenará un array de estructuras de tipo ***vaGeoLod*** en el que se almacenan informaciones sobre las distintas formas de representación de un actor virtual, y sus márgenes de representación. La estructura *vaGeoLod* tiene el siguiente aspecto:

```
typedef struct vaGeoLodStruct{
    float distMin, distMax;     // Margen de distancias entre las que se aplica.
    float numVertices;          // Numero de vértices.
    float drawCost;             // Estimacion del coste de dibujado en microSegundos.
}vaGeoLod;
```

En el campo ***numGeoLods*** del nodo *Actor* se almacena el número de diferentes niveles de detalle empleados para representar a un actor, y en el campo ***geoLods***, un array de punteros a estructuras de tipo *vaGeoLod*. Esta información es empleada para realizar estimaciones del coste de dibujado de los actores virtuales, las cuales serán empleadas por los métodos de gestión de culling y LOD.

Los actores virtuales emplearan una gestión de culling distinta de la estándar, en la que la mayoría de los casos no será necesario realizar comprobaciones de CULL con las *bounding spheres* de las distintas geometrías que componen internamente al actor. El nodo *Actor* almacenará en su interior dos *bounding spheres*, la primera de ellas, de nombre ***sklsrootBsph*** (*SklsRoot Bsphere*), estará centrada en el *Skeletons Root* del actor, y tendrá un radio suficiente como para englobar al actor virtual completo independientemente de la postura que éste adopte. La segunda, de nombre ***refpointBsph*** (*Reference Point Bsphere*), es capaz de englobar al actor virtual y su movimiento. La forma en la que se realiza la gestión del culling de los actores virtuales será mostrada en detalle en un capítulo posterior.

4.4.3.3 Nodo *Actor* como elemento gestor de comportamiento.

La gestión del comportamiento de los actores podría ser realizada desde un gestor general, que se encargue a cada frame de actualizar los valores de los grados de libertad de todos los actores de la escena. Sin embargo, resulta mucho más adecuado que cada actor virtual gestione su propio comportamiento de forma autónoma. En este sentido el nodo *Actor* incorpora un conjunto de campos destinados a almacenar las funciones de gestión de su comportamiento (de nombre ***refpointBehaviouFunc***, ***sklsrootBehaviourFunc*** *y* ***dofsBehaviourFunc***), y los datos que puedan ser necesarios para realizar dicha gestión (***behaviourData***).

Las funciones empleadas en la gestión del comportamiento del actor tienen la siguiente sintaxis:

```
typedef void   (*vaActorBehaFunc)(        vaActor *act);
```

El hecho de que dicha función reciba un puntero al nodo *Actor* le proporciona un acceso a la información almacenada en el campo *behaviourData*, a la distancia del actor a la cámara, los valores actuales de los grados de libertad, o incluso uso la posición espacial de los distintos puntos de articulación. Dichos valores

pueden ser empleados para realizar sus cálculos internos, que suelen finalizar con la modificación de los valores de *refpointPos*, *refpointHpr*, *sklsrootPos*, *sklsrootHpr* y de los grados de libertad. El hecho de que estas funciones sean invocadas por el propio nodo *Actor* facilita que puedan estar relacionada con el proceso de culling, y también con el proceso de gestión de nivel de detalle.

4.4.4 Estructura de datos.

Una vez presentados algunos de los aspectos básicos del nodo *Actor*, resulta interesante observar su estructura de datos:

```
typedef struct vaActorStruct{
   char    name[VA_MAX_NAMELENGTH];

   // Informacion sobre el Reference Point.
   float   refpointPos[3];              // Posicion.
   float   refpointHpr[3];              // Orientacion                Rz->Ry->Rx.
   float   refpointMat[16];             // Matriz compuesta de refpointPos y refpointHpr.
   int      refpointMatDirty;           // Flag que indica si refpointMat es valida.

   // Informacion sobre  Skeletons Root.
   float   sklsrootPos[3];              // Posicion.
   float   sklsrootHpr[3];              // Orientacion                Rz->Ry->Rx.
   float   sklsrootMat[16];             // Matriz compuesta de sklsrootPos  y sklsrootHpr.
   int      sklsrootMatDirty;           // Flag que indica si sklsrootMat es valida.

   float   actorMat[16];                // Matriz total aplicada por el nodo actor.

   float   refpointAbsMat[16];          // Matriz absoluta del Reference Point.
   float   sklsrootAbsMat[16];          // Matriz absoluta del Skeletons Root.

   float    * conf;                     // Array de valores de sus grados de libertad.
   vaSkeleton ** lSkls;                 // Array de punteros a sus nodos Skeleton.

   //- Informaciones para la gestion de Culling y LOD.
   float    refpointBsph[4];            // Bounding sphere del actor y su movimiento.
   float    sklsrootBsph[4];            // Bounding sphere del cuerpo del actor.

   int      numGeoLods;                 // Informacion sobre sus niveles de detalle geometricos.
   vaGeoLod  **geoLods;

   int      intBsphsCullOn;             // Flag de activacion del culling con las internal bspheres.
   int      refpointBsphIn;
   int      sklsrootBsphIn;

   float    eyeDistance;                // Distancia a la camara.
   float    priority;                   // Prioridad del actor en la escena 0<priority<1.

   //- informaciones empleadas en la gestion del comportamiento.
   void *                behaviourData;
   vaActorBehaFunc   refpointBehaviourFunc;
   vaActorBehaFunc   sklsrootBehaviourFunc;
   vaActorBehaFunc   dofsBehaviourFunc;

   vaActclass *actclass;                // Puntero a su Actor Class.
   void    *sgNode;
   void    *extensionSlots[VA_MAX_EXTENSIONS];
}vaActor;
```

El significado de la mayoría de los campos contenidos en la estructura de datos del nodo *Actor* ha sido descrito en los apartados anteriores, solamente quedan sin explicar el significado de los campos *name*, *sgNode*, *actClass* y *extensionSlots.* El campo ***name*** almacena el nombre del actor, y ha de ser un nombre único. El campo ***sgNode*** almacena un puntero su nodo del grafo de escena. El campo ***actClass*** es un puntero a una estructura de datos *vaActclass*, que es la encargada de almacenar diversos tipos de informaciones de alto nivel que pueden ser empleadas por un determinado grupo de actores con características similares. Esta estructura de datos será presentada en el siguiente capítulo. El campo ***extensionsSlots*** proporciona un método para poder ampliar el contenido de un nodo *Actor*, y también será explicado en detalle en siguiente capítulo.

4.4.5 Gestión de las transformaciones. Estrategias de reducción de coste computacional.

El procesado de un nodo *Actor* tienen dos costes principales, el primero de ellos es el cálculo de los valores que definen la posición del actor en la escena (vectores *refpointPos*, *refpointHpr*, *sklsrootPos* y *sklsrootHpr*), el cual es llevado mediante las funciones *refpontBehaviourFunc* y *sklsrootBehaviourFunc*, y el segundo, la generación de las distintas matrices de transformación (*refpointMat*, *refpointAbsMat*, *sklsrootMat* y *sklsrootAbsMat*) a partir de dichos valores. La generación de estas matrices es realizada de forma interna en el nodo *Actor,* y puede tener un coste computacional considerable, es este apartado se analiza la forma de realizar estos cálculos de una forma adecuada.

Para analizar de una forma independiente el coste de generación de la matriz de transformación aplicada por un actor vamos a suponer que disponemos de los 12 parámetros que definen la posición del actor en la escena (vectores *refpointPos*, *refpointHpr*, *sklsrootPos* y *sklsrootHpr*), y que son procesados para obtener las matrices que definen la posición relativa y absoluta del *Reference Point* (*refpointMat* y *refpointAbsMat*), y del *Skeletons Root* (*sklsrootMat* y *sklsrootAbsMat*). En este contexto, las secuencia de operaciones a realizar serían las siguientes:

```
matRefpointPos    = generateTranslateMatrix( refpointPos[X], refpointPos[Y], refpointPos [Z] );
matRefpointRz     = generateRotzMatrix( refpointHpr[ 0] );
matRefpointRy     = generateRotyMatrix( refpointHpr [1] );
matRefpointRx     = generateRotxMatrix( refpointHpr [2] );
refpointMat       = matRefpointPos  × matRefpointRz × matRefpointRy  × matRefpointRx;
refpointAbsMat    = getCurrentModelMatrix()  × refpointMat;

matSklsrootPos    = generateTranslateMatrix( sklsrootPos[X], sklsrootPos[Y], sklsrootPos[Z]);
matSklsrootRz     = generateRotzMatrix( sklsrootHpr[0] );
matSklsrootRy     = generateRotyMatrix( sklsrootHpr[1] );
matSklsrootRx     = generateRotxMatrix( sklsrootHpr[2] );
sklsrootMat       = matSklsrootPos × matSklsrootRz × matSklsrootRy  × matSklsrootRx;
actorMat          = sklsrootMat × refpointMat;
sklsrootAbsMat    = getCurrenModelMat() × actorMat;
```

Generación directa de las matrices de tranformación de un nodo *Actor*.

Para calcular las matrices del *Reference Point* es necesaria la generación de 4 matrices de transformación básicas (con un coste de 6 operaciones seno/coseno), y realización de 4 multiplicaciones de matrices de 4x4 (con un coste de 256 multiplicaciones y 192 sumas).

De forma similar, para la obtención de las matrices del *Skeletons Root* sería necesaria la generación de 4 matrices de transformación básicas (otras 6 operaciones de seno/coseno), y la realización de 5 multiplicaciones de matrices 4x4 (320 multiplicaciones y 240 sumas).

Para evaluar el coste del procesado de los actores virtuales resulta adecuado considerar por separado el coste de generación de las matrices del *Reference Point* (al que nos referiremos partir de ahora como *costeRefPointMats)*, y el coste de generación de las matrices del *Skeletons Root (*al que nos referiremos a partir de ahora como *costeSklsRootMats)*. También, para simplificar los cálculos, vamos a suponer, al igual que se hizo con los nodos *Skeleton*, que tanto la suma de dos números flotantes, como la operación de seno o coseno tienen un coste equivalente a la realización de una multiplicación de números flotantes (abreviado como FMULT). Empleando esta nomenclatura, los costes de generación de las matrices de un nodo *Actor* serían los siguientes:

costeSklsRootMats = 6 ops. seno/coseno, 256 mults y 192 sumas = 454 FMULTs.
costeRefPointMats = 6 ops. seno/coseno, 320 mults y 240 sumas = 566 FMULTs

La operación de cálculo de las matrices de transformación de un nodo Actor no es tan habitual como la operación de cálculo de las matrices de un nodo Skeleton. Aún así, tienen un coste computacional considerable. En los siguientes apartados se presentan distintas estrategias empleadas acelerar la generación de estas matrices.

Los nodos *Actor* necesitan conocer siempre las posiciones absolutas del *Reference Point* y el *Skeleton Root,* (puesto que son empleadas en los procesos de culling y selección de niveles de detalle), esto fuerza a que las multiplicaciones de matrices tengan que ser realizadas siempre en la CPU, y no se pueda recurrir al hardware de multiplicación de matrices de forma similar a como se hacia en el caso de los nodos *Skeleton* (Apartado 4.3.6.4).

4.4.5.1 Cálculo optimizado de la Matrices de transformación.

Tanto la matriz *refpointMat* como la matriz *sklsrootMat* descritas anteriormente, son generadas a partir de la concatenación de la siguiente secuencia de matrices de transformación básicas:

$$\underset{\textbf{Translation}}{\begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}} \times \underset{\textbf{rot (eje Z)}}{\begin{bmatrix} \cos Rz & -sinRz & 0 & 0 \\ sinRz & \cos Rz & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} \times \underset{\textbf{rot(eje Y)}}{\begin{bmatrix} \cos Ry & 0 & sinRy & 0 \\ 0 & 1 & 0 & 0 \\ -sinRy & 0 & \cos Ry & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} \times \underset{\textbf{rot(eje X)}}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos Rx & -sinRx & 0 \\ 0 & sinRx & \cos Rx & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}$$

Al igual que ocurría en el caso de los nodos *Skeleton*, cualquiera de las matrices básicas tienen al menos 10 valores que son 0 y al menos 2 elementos con valor 1, operar directamente sobre estas matrices sin tener en cuesta esta característica supondría que aproximadamente un 75% de las operaciones se realizarían de forma innecesaria. Este inconveniente puede ser subsanado realizando una concatenación algebraica de dichas matrices, cuyo resultado sería la siguiente expresión:

Matriz de translation-heading-pitch-roll

$$\begin{bmatrix} \cos Rz * \cos Ry & \cos Rz * sinRy * sinRx - sinRz * \cos Rx & \cos Rz * sinRy * \cos Rx + sinRz * sinRx & Tx \\ sinRz * \cos Ry & sinRz * sinRy * sinRx + \cos Rz * \cos Rx & sinRz * sinRy * \cos Rx - \cos Rz * sinRx & Ty \\ -sinRy & \cos Ry * sinRx & \cos Ry * \cos Rx & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Esta expresión aún puede ser representada de una forma más sencilla haciendo que los cálculos trigonométricos, y algunas operaciones que se repiten, sean realizadas en una fase previa a la generación de la matriz. La expresión final sería:

Matriz simplificada de translation-heading-pitch-roll-tx

$A = cosRz;\ B = cosRy;\ C = cosRx$

$D = sinRz;\ E = sinRy;\ F = sinRx$

$AC = A*C;\quad AF = A*F;$

$CD = C*D;\ DF = D*F$

$$\begin{bmatrix} A*B & AF*E - CD & AC*E + DF & Tx \\ D*B & DF*E + AC & CD*E - AF & Ty \\ -E & B*F & B*C & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Esta expresión será implementada en una función de nombre *computeThprMatrix(),* la cual tendrá un coste de 6 operaciones de calculo de senos y cosenos, 12 multiplicaciones y 4 sumas. Al coste de generación de esta matriz lo identificaremos como *costeThprMat* y tiene un valor de 22 FMULTs.

Si modificamos la secuencia de código mostrado con anterioridad para que emplee esta nueva función se obtiene el siguiente bloque de código al que nos referiremos con el nombre de "*Optimización A*":

```
refpointMat      = computeThprMatrix(      refpointPos[0], refpointPos[1], refpointPos[2],
                                           refpointHpr[0], refpointHpr[1], refpointHpr[2]);
refpointAbsMat   = getCurrentModelMatrix() × refpointMat;

sklsrootMat      = computeThprMatrix(      sklsrootPos[0], sklsrootPos[1], sklsrootPos[2],
                                           sklsrootHpr[0], sklsrootHpr[1],  sklsrootHpr[2]);
actorMat         = refpointMat ×  sklsrootMat;
sklsrootAbsMat   = getCurrentModelMatrix()×  actorMat;
```

Optimización A. Incorporación del cálculo optimizado de las matrices relativas.

En la figura anterior, hay dos tipos de operaciones, la operación *computeThprMatrix* y la multiplicación de matrices de 4x4. Si expresamos estas multiplicaciones con la abreviatura *costeMultMat,* de forma indicada en la *Tabla 4-3*, el coste generación de las matrices de un nodo *Actor* podría ser expresada como:

costeRefPointMats = costeThprMat + costeMultMat= 134 MULTs.

costeSklsRootMats = costeThprMat + 2*costeMultMat = 246 FMULTs.

4.4.5.2 Empleo de funciones especiales para la multiplicación de matrices afines.

En la secuencia de operaciones mostrada en bloque de código correspondiente con la ***¡Error!No se encuentra el origen de la referencia.***, se realizan tres multiplicaciones de matrices de 4x4, estas matrices tienen su última fila formada por tres ceros y un 1, esta característica puede ser aprovechada para sustituir la multiplicación genérica de matrices 4x4 por una función especifica para la multiplicación de matrices afines, (*composeAffineMatrix*). El coste de esta operación (*costeMultAfin*), ya ha sido descrito para estimar el coste computacional de procesado del nodo Skeleton, y es de 63 FMULTs. La secuencia de operaciones a realizar empleando esta nueva función será nombrada como *"Optimización B"* , y es la siguiente:

```
refpointMat      = computeThprMatrix(      refpointPos[0], refpointPos[1], refpointPos[2],
                                           refpointHpr[0], refpointHpr[1],  refpointHpr[2]);
refpointAbsMat   = composeAffineMatrix( getCurrentModelMatrix() , refpointMat);

sklsrootMat      = computeThprMatrix( sklsrootPos[0],  sklsrootPos[1], sklsrootPos[2],
                                      sklsrootHpr[0], sklsrootHpr[1],  sklsrootHpr[2]);
actorMat         = composeAffineMatrix ( refpointMat, sklsrootMat);
sklsrootAbsMat   = composeAffineMatrix ( getCurrentModelMatrix(),  actorMat);
```

Optimización B. Incorporación de las funciones de multiplicación de matrices afines.

El coste de generación de las matrices del nodo *Actor* podría ser expresado ahora como:

costeRefPointMats = costeThprMat + costeMultAfin = 86 FMULTs.

costeSklsRootMats = costeThprMat + 2×costeMultAfin = 148 FMULTs

4.4.5.3 Bloqueo de cálculos por coherencia temporal.

Si bien, en principio es necesario asegurar que los valores almacenados en las matrices del nodo *Actor,* se actualicen a una frecuencia elevada (por ejemplo 50 frames/segundo), existirán muchos casos prácticos en los que los valores almacenados en dichas matrices no presentan ninguna modificación respecto a instantes anteriores, y, por tanto, su recálculo estaría consumiendo recursos de una forma innecesaria. Para evitar estas perdidas de eficiencia, los nodos *Actor* almacenan las matrices *sklsrootMat*, *refpointMat* y *actorMat,* e incorporan los flags ***sklsrootMatDirty***, y ***refpointMatDirty***, que permiten determinar cuando los recálculos son realmente necesarios

Para comprender el funcionamiento de estos flags veamos un ejemplo práctico: Supongamos un patinador está sentado en el borde de la pista de patinaje descansando (totalmente estático, sus matrices *refpointMat* y

sklsrootMat permanecen fijas). En un momento determinado se levanta, y comienza a realizar ejercicios de calentamiento, pero sin moverse de su sitio (el hecho de que su posición no cambie hace que *refpointMat* permanezca intacta, sin embargo sus ejercicios de calentamiento harían que el contenido de la matriz *sklsrootMat* estuviese cambiando). Supongamos ahora que comienza a moverse sobre la pista tomando velocidad (esto implicaría variaciones en los dos tipos de transformaciones), y por último supongamos que una vez ha tomado velocidad deja de mover su cuerpo y se desplaza por la pista usando tan sólo la inercia adquirida, en este último caso tan sólo se estarían sufriendo modificaciones sobre la matriz *refpointMat*. Los estados de los flags son mostrados en la siguiente tabla:

Estado del patinador	**sklsrootMatDirty**	**refpointMatDirty**
Patinador sentado descansando.	FALSE	FALSE
Patinador realizando ejercicios de calentamiento	FALSE	TRUE
Patinador tomando velocidad.	TRUE	TRUE
Patinador deslizándose aprovechando su inercia.	TRUE	FALSE

El contenido del flag *actorMatDirty* suele ser en la mayoría de los casos el resultado de realizar una operación "o" lógica entre los flags *sklsrootMatDirty* y *refpointMatDirty*.

```
int actMatDirty = refpointMatDirty || sklsrootMatDirty;
if ( refpointMatDirty ) {
        refpointMat             = computeThprMmatrix( refpointPos[0], refpointPos[1], refpointPos[2],
                                                      refpointHpr[0],  refpointHpr[1], refpointHpr[2]);
        retpointMatDirty        = FALSE;
}
refpointAbsMat      = composeAffineMatrix( getCurrentModelMatrix(), refpointMat);

if ( sklsrootMatDirty ) {
        sklsrootMat   = computeThprMatrix(      sklsrootPos[0], sklsrootPos[1], sklsrootPos[2],
                                                sklsrootHpr[0], sklsrootHpr[1],  sklsrootHpr[2]);
        sklsrootMatDirty = FALSE;
}
if (actMatDirty){
        actorMat                = composeAffineMatrix(refpointMat, sklsrootMat);
}
sklsrootAbsMat      = composeAffineMatrix( getCurrentModelMatrix(),  actorMat);
```

Optimización C. Incorporación de comprobaciones de coherencia temporal.

El coste de la generación de las matrices en este caso depende del *factor de coherencia temporal*, el cual indica cuanto tiempo permanece una de las matrices sin alteración: un valor para dicho factor de 1 indica que se actualiza a cada frame, y un valor de 0 indica que permanece sin modificación de forma contínua. Si se expresa dicho factor con la abreviatura *fCoherencia*, los costes de generación de las matrices podría ser expresado como:

$$costeSklsRootMats = fCoherencia \times costeTHPRm + costeMultAfin$$

$$costeRefPointMats = fCoherencia \times (costeTHPRm + costeMultAfin) + costeMultAfin$$

Si esto se expresa en FMULTs.

costeSklsRootMats = (fCoherencia × 22 + 63) FMULTs

costeRefPointMats = (fCoherencia × 85 + 63) FMULTs

4.4.5.4 Estimación de la mejora.

Observando de forma conjunta las mejoras introducidas por cada optimización, se obtiene la siguiente tabla:

	costeRefPointMats	costeSklsRootMats	costeTotal.
caso Original	454 FMULTs	566 FMULTs	1020 FMULTs
Optimización A	134 FMULTs	246 FMULTs	380 FMULTs
Optimización B	86 FMULTs.	148 FMULTs	234 FMULTs
Optimización C (f = 0.7)	78 FMULTs	122 FMULTs	200 FMULTs
Optimización C (f = 0.2)	67 FMULTs	80 FMULTs	147 FMULTs

Como se puede observar, en el caso de la optimización por coherencia temporal se han empleado dos valores distintos para el *factor de coherencia temporal*. La utilización de las optimizaciones consigue que la generación de las matrices de un nodo *Actor* se realicen entre 5 y 7 veces más rápido.

Los costes de generación de las matrices del *Reference Point* y del *Skeletons Root,* con y sin optimizaciones son los mostrados en la siguiente tabla:

	Situación Original.	Con todas las optimizaciones.
costeSklsRootMats	454 FMULTs	63 + fCoherencia*22 FMULTs
costeRefPointMats	566 FMULTs	63 + fCoherencia*85 FMULTs

Estas expresiones serán empleadas más adelante para realizar una evaluación del coste total de procesado de una escena con múltiples actores virtuales.

4.4.6 Procesado de un nodo Actor.

A diferencia del nodo *Skeleton*, cuyo procesado consistía básicamente obtener y aplicar de una forma eficiente una matriz de transformación, el nodo *Actor* actúa como centro de control del actor virtual, y por tanto, requiere un control mucho más complejo. Toda la gestión del nodo *Actor* es realizada en una función *actorMainFunc()*, que es llamada en el momento en el que el recorrido del *grafo de escena* alcanza uno de estos nodos. Dicha función está encargada de:

- Calcular y aplicar los valores que definen la posición del actor en el espacio.
- Calcular la distancia del *Skeletons Root* al punto de vista, esta distancia será empleada para la gestión del nivel de detalle y el culling.
- Calcular y aplicar los valores de sus grados de libertad.
- Comprobar la relación de la *refpointBsph* y la *sklsrootBsph* con el frustum, y bloquear los cálculos que no sean necesarios.

En los apartados anteriores se ha mostrado que la gestión del comportamiento estaba dividida en tres funciones, y también que se emplea un método especial de culling en el que habrá tres tipos de comprobaciones. En la función principal de procesamiento de un actor las comprobaciones de CULL aparecen mezcladas con la gestión del comportamiento, de tal modo que algunas de las funciones de comportamiento no son llamadas si el culling determina que no es necesario.

En dicha función también se determina si se ha de procesar el subgrafo de escena que depende de ese nodo *Actor,* y en caso afirmativo, se establece si se han de tener o no en cuenta las *bounding spheres* de los distintos nodos existentes en su subgrafo. Este control se realiza a través del valor retornado por la función cuyo significado es el siguiente:

0 => Actor fuera del frustum. No continuar haciendo nada con el.

1 => Actor dentro del frustum. Procesar el trozo de *grafo de escena* que depende de este nodo *Actor*, pero sin realizar ningún tipo de comprobación de CULL con sus nodos internos.

2 => Actor dentro del frustum. Calcular intersecciones con *bspheres* de sus nodos internos.

Internamente dicha función tiene la siguiente a forma:

```
int actorMainFunc(vaActor *act){
	(*act->refpointBehaviourFunc)( act );			//-Se calculan los valores de refpointPos y refpointHpr.
	int actMatDirty = act->refpointMatDirty || act->sklsrootMatDirty;
	if ( act->refpointMatDirty ) {
	    act->refpointMat = computeThprMatrix( act->refpointPos[0], act->refpointPos[1],  act->refpointPos[2],
	                                          act->refpointHpr[0],  act->refpointHpr[1],  act->refpointHpr[2]);
	    act->retpointMatDirty = FALSE;
	}
	act->refpointAbsMat = composeAffineMatrix(getCurrentModelMatrix( ) , act->refpointMat);
	if  (checkBsphOutFrustum(act->refpointAbsMat, act->refpointBsph))
	    return 0;

	(*act->sklsrootBehaviourFunc)( act );	//-Se calculan los valores de sklsrootPos y sklsrootHpr.
	if (act->sklsrootMatDirty ) {
	    act->sklsrootMat = computeThprMatrix(  act->sklsrootPos[0], act->sklsrootPos[1], act->sklsrootPos[2],
	                                           act->sklsrootHpr[0], act->sklsrootHpr[1],  act->sklsrootHpr[2]);
	    act->sklsrootMatDirty = FALSE;
	}
	if (actMatDirty){
	    act->actorMat	= composeAffineMatrix(act->refpointMat, act->sklsrootMat);
	}
	act->sklsrootAbsMat   = composeAffineMatrix( getCurrentModelMatrix(), act->actorMat);
	if (checkBsphOutFrustum( act->sklsrootAbsMat,  act->sklsrootBsph)
	    return 0;				//-Actor fuera del frustum.

	setCurrentModelMatrix(act-> sklsrootAbsMat );
	(*act->dofsBehaviourFunc)( act);
	if ( checkWithInternalBspheres(act)  )
	    return 2;				//-Actor dentro del frustum. check internal bpheres.
	else
	    return 1;				//-Actor dentro del frustum, NO check internal bspheres.
}
```

La gestión del culling también puede suponer un ahorro en la gestión del comportamiento, puesto que como se puede observar, la función *sklsrootBehaviourFunc*, sólo es llamada si la *refpontBsph* está dentro del

frustum, y de forma similar, la función *dofsBehaviourFunc* solamente es llamada si la *sklsrootBsph* está dentro del *frustum*. La función *refpointBehaviourFunc* se aplica sobre todos los actores a los que llegue el recorrido del *grafo de escena*. Puede que un actor no sea procesado por estar en una sección de *grafo de escena* que ha sido eliminado por el CULL tradicional (por ejemplo un actor que estuviese dentro de un edificio que se encuentra totalmente fuera del *frustum*).

La función *sklsrootBehaviourFunc*, es invocada en un momento en el que ya se conoce la posición absoluta del *Reference Point* (r*efpointAbsMat*), de este modo conoce su distancia a la cámara, que puede ser empleada para realizar una gestión de nivel de detalle de comportamiento en el interior de dicha función.

De un modo similar, la función *dofsBehaviourFunc* es llamada en un momento en el que ya se conoce la posición absoluta del *Skeletons Root* (*sklsrootAbsMat*), y, por tanto, se puede emplear el valor de la distancia a la cámara para hacer que dicha función también pueda realizar una gestión del nivel de detalle del comportamiento.

La función *checkWithInternalBspheres(),* determina si conviene realizar culling con las internal bspheres dependiendo del coste de dibujado (puede ser obtenido a partir de la tabla de *geoLods*), y de una estimación del coste de realización del CULL (que puede ser puede ser estimado a partir del número de nodos *Skeleton* que componen al actor, y su topología).

4.5 Conclusiones.

El empleo de una librería basada en un grafo de escena, constituye la forma más estandarizada de implementar una aplicación de simulación. El diseño de los grafos de escena actuales hace que estén orientados hacia la visualización de grandes bases de datos, que usualmente contienen objetos estáticos (terreno, edificios, etc.), u objetos con movimientos sencillos (básicamente vehículos), no estando preparados para la correcta definición y gestión de escenas que incluyan varios actores virtuales.

En este capítulo se han presentado dos nuevos tipos de nodos, especialmente diseñados para integración de actores sintéticos en aplicaciones de simulación en tiempo real. Estos dos nodos son llamados ***Actor*** y ***Skeleton*****,** proporcionan al usuario un control de alto nivel sobre los actores virtuales, y realizan una gestión a bajo nivel que minimiza su coste computacional. El *nodo Actor* actúa como nodo raíz de una jerarquía de nodos *Skeleton,* encargada de definir la estructura articulada del actor. Cada nodo *Skeleton* define un punto de articulación del actor, como por ejemplo el codo, la rodilla o el cuello. La postura del actor en un momento dado depende de las transformaciones afines aplicadas por los nodos *Skeleton* en el grafo de escena; la ubicación del actor en la escena depende de las transformaciones aplicadas por el nodo *Actor*. La transformación aplicada por cada *Skeleton* es controlada mediante un conjunto de variables, asociados con los grados de libertad, que son almacenados en el nodo *Actor* correspondiente (no en el propio nodo *Skeleton*). El nodo *Actor* contiene una tabla de parámetros que definen la postura, y los nodos *Skeleton* acceden a dicha lista, siempre que necesitan conocer los valores actuales de dichos parámetros. De este modo, la modificación de la postura del actor es llevada a cabo mediante la simple actualización de los parámetros almacenados en la tabla del nodo *Actor*. Este tipo de relación, que no es común en un grafo de escena, hace que la complejidad jerárquica del actor virtual resulte transparente al usuario.

El diseño de estos nuevos nodos ha sido hecho desde una óptica multiplataforma, partiendo de la abstracción de que estos nodos derivan de un *nodo Grupo* tradicional, existente en cualquier grafo de escena tradicional. Esto permite la integración de dichas estructuras en cualquier grafo de escena que cumpla unos requisitos mínimos, y facilita la relación entre los actores virtuales y el resto de los elementos de la escena, haciendo por ejemplo que un nodo genérico del grafo de escena pueda depender de un nodo *Actor* o *Skeleton* (por ejemplo una herramienta), o que un actor virtual pueda ser insertado en cualquier punto de la escena (por ejemplo en el interior de un vehículo).

El punto crítico en la integración de actores virtuales en una aplicación simulación es, sin duda, el diseño de su estructura articulada, más aún si se buscan como objetivos la estandarización, la sencillez de uso, y la eficiencia computacional. En este capítulo se ha hecho un análisis exhaustivo sobre cual sería el diseño óptimo de la estructura articulada de un actor virtual. En este sentido, se ha sustituido la organización clásica en los sistemas de animación 3D formada por parejas Segmento-Junta (el *Segmento* hace referencia a hueso que separa dos puntos de articulación, actuando también como elemento de interfaz; la *Junta* define el punto de articulación y sus grados de libertad), por un elemento único (*nodo Skeleton*) que almacena los grados de

libertad de una articulación, un offset a la articulación de la que depende, y puede, también, actuar como elemento de interfaz.

El objetivo básico del nodo Skeleton es proporcionar un método que permita definir una estructura articulada de una forma estándar. Después de un análisis detallado de todas las opciones existentes, se ha optado por: emplear un criterio de ejes *"right-handed"* con eje Z vertical; emplear un sistema de codificación de rotaciones basado en ángulos de *Euler* aplicados en un orden Rz->Ry->Rx; hacer que el nodo *Skeleton* esté formado por 4 grados de libertad, tres rotacionales (Rz, Ry, Rx), y uno adicional de traslación (Tx) y hacer que los cálculos intermedios se realicen mediante matrices de transformación "*row-major*" que se aplican de forma concatenada. También se han analizado los problemas relacionados con la utilización de Euler (*Gimbal-lock* e interpolación) y se han propuesto soluciones. Se ha presentado la estructura del nodo *Skeleton,* y se han aislado los distintos tipos de operaciones que intervienen es su procesado, realizando una evaluación teórica del coste de procesado, evidenciando el cuello de botella existente en el ámbito de las multiplicaciones de matrices, y mostrando la conveniencia de utilización de hardware gráfico con esta capacidad. Se han propuesto varias estrategias que consiguen mejorar su rendimiento computacional en un factor que va entre 30 y 60 en el caso de disponer de hardware de multiplicación de matrices, y entre 9 y 12 en caso contrario.

Respecto al nodo Actor, se han mostrado sus dos principales cometidos: actuar como sistema de referencia base para todos los nodos que componen el actor virtual, y actuar como centro de control de todas las acciones del actor virtual. Desde la primera óptica, se ha mostrado la conveniencia de hacer que el actor emplee dos grupos de transformaciones, el primero de ellos definiendo la posición de un punto denominado *Reference Point* normalmente asociado al centro de masas del actor (si bien puede ser empleado para otros propósitos), y el segundo que define la posición del *Skeletons Root*, un punto fijo localizado en algún punto de la estructura ósea del actor, y que actúa como sistema de referencia base para el resto de su estructura articulada. Desde el punto de vista del actor como centro de control se han mostrado la forma en el que el usuario puede modificar la postura del actor, o conocer la posición que cualquiera de sus articulaciones mediante actuaciones directas sobre el nodo *Actor*. También se ha mostrado la forma en la que actúa como soporte para métodos especiales de gestión de culling y nivel de detalle, y también como centro de gestión del comportamiento. Al igual que en el caso del nodo *Skeleton*, se ha mostrado su estructura de datos, se ha aislado las distintas operaciones que intervienen en su procesado, y se han propuesto varias estrategias encaminadas a optimizar su coste computacional. También se ha mostrado la forma en las funciones de gestión de comportamiento y culling actúan entremezcladas para optimizar el rendimiento.

Los nodos *Actor* y *Skeleton* propuestos en este capítulo proporcionan una forma sencilla y eficiente de ampliar la funcionalidad de los grafos de escena actuales. El uso de este tipo de estructuras debería contribuir a incrementar el número de aplicaciones de informática gráfica en tiempo real que incorporen actores virtuales.

Capítulo 5. Estructura de Clases de Actores.

5.1 Índice.

5.2 Introducción.

En los apartados anteriores se han analizado cuales serían las estrategias más adecuadas para conseguir integrar a los actores virtuales dentro de un *grafo de escena* de un modo eficiente. Como resultado se han definido dos nuevos tipos de nodos, y se ha expuesto en detalle cual sería su misión y su contenido. La utilización de estos dos nuevos tipos de nodos permite definir de una forma eficiente la estructura esquelética del actor virtual, y contienen información suficiente para definir totalmente su apariencia geométrica. Sin embargo, los actores son entidades complejas compuestas de muchos aspectos a parte del propiamente geométrico, siendo necesaria una zona en la que poder almacenar informaciones relativas a su cinemática, dinámica, métodos de desplazamiento, controles basados en inteligencia artificial etc. En un primer momento se puede pensar que dichas informaciones pueden ser almacenadas directamente en los nodos *Actor* o *Skeleton* del *grafo de escena,* sin embargo, existen varios factores hacen que esta aproximación no resulte adecuada:

- En general los nodos de un *grafo de escena* almacenan tan sólo información que afecte directamente al dibujado (bien sea aplicando una transformación afín como los nodos DCS, una selección entre los nodos hijos como los nodos SWITCH o LOD, etc.). La información de tipo no geométrico asociada a un actor virtual puede llegar a tener un volumen considerable, y el *grafo de escena* no es el lugar adecuado para su almacenamiento.

- Las informaciones de alto nivel asociadas a los actores sintéticos pueden ser compartidas por varios actores existentes en la misma escena, así por ejemplo un método de alto nivel para caminar podría ser utilizado por todos los actores de tipo humanoide existentes en la simulación. Resulta pues adecuada la existencia de una estructura de más alto nivel, que se encargue de almacenar los atributos de tipo general, que son comunes a una determinada "especie" de actores.

- Es necesario que las informaciones de alto nivel asociadas a los actores virtuales puedan ser ampliadas. Uno de los objetivos de este trabajo, es proporcionar un substrato básico que pueda absorber fácilmente futuras ampliaciones. Los nodos *Actor* y *Skeleton* proporcionan un soporte estable para definir las características geométricas del actor, una estructura de más alto nivel será ha de ser la encargada de absorber todas las futuras ampliaciones.

- La existencia de entornos de simulación complejos que puedan contener una cantidad enorme de actores (por ejemplo la simulación de un conjunto de calles con personas caminando, o un edificio con múltiples habitaciones habitadas), necesita emplear descripciones de tipo macroscópico sobre las poblaciones de actores. Dichas descripciones macroscópicas han de poder ser traducidas a representaciones de actores individuales, en el momento en el que sea necesaria su visualización. El empleo de este tipo de técnicas requiere la creación y eliminación de actores en tiempo de ejecución.

Es necesario algún tipo de estructura que permita realizar este tipo de operación de forma rápida, y manejando una cantidad mínima de información.

En este trabajo se propone la utilización de una estructura independiente del *grafo de escena* que almacene todas las informaciones de alto nivel de los actores virtuales. Esta estructura, a la que se le da el nombre de ***Clase de Actor*** almacena todas las informaciones de tipo lógico propias de una ¨especie¨ (por ejemplo de la especie humano) y de la cual los actores concretos existentes en el *grafo de escena* son considerados como instancias. Los actores pertenecientes a una determinada *Clase de Actor* comparten una misma estructura topológica, y también sus métodos de alto nivel (mirar, caminar, etc.). A lo largo de este trabajo, se nombrará como ***vaActclass*** a la estructura de datos empleada para almacenar la información referente a una *Clase de Actor*.

En una escena de simulación tradicional toda la información suele ser almacenada en el propio *grafo de escena.* En este trabajo, a diferencia, se propone un tipo de escena 3D en la que la información es almacenada en dos zonas bien diferenciadas: La información directamente relacionada con el dibujado de la geometría es almacenada también en un *grafo de escena*, pero la información de alto nivel relacionada con los actores es almacenada de forma independiente por medio de una **lista de estructuras de tipo *vaActclass*.** La *Figura 5-1* representa esta forma de distribuir la información: ciertos datos son almacenados en los nodos *Actor* y *Skeleton* bajo una estructura de *grafo de escena* tradicional, y otros son almacenados en una lista de *Clases de Actores*. Se puede observar como las nodos *Actor* del *grafo de escena* acceden a sus correspondientes estructuras de tipo v*aActclass* para extraer ciertos datos o utilizar ciertos métodos.

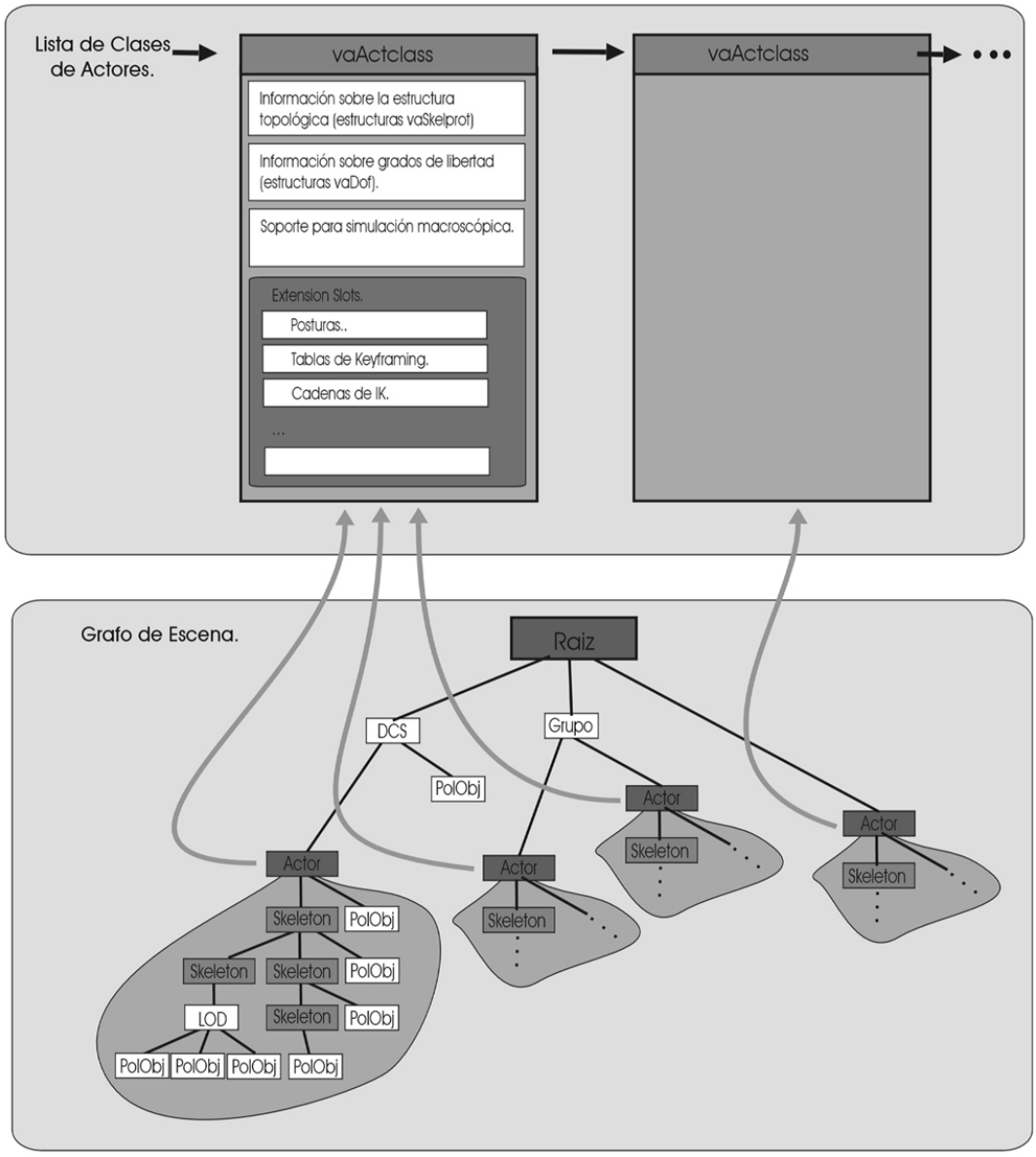


Figura 5-1. Escena compuesta por una lista de *Clases de Actores* y un *Grafo de Escena* con nodos *Actor* y *Skeleton*.

En los siguientes apartados se realiza una descripción del contenido de una estructura *Clase de Actor*, mostrando, además, su capacidad para actuar como soporte para ampliaciones, y para la gestión de simulación macroscópica.

5.3 Información contenida en una *Clase de Actor*. Aspectos generales.

Cada uno de los elementos de la lista de clases de actores almacena las informaciones de tipo general y de alto nivel relativas a una especie concreta de actores. Cada actor individual almacena sus características específicas (básicamente su apariencia geométrica y los valores que definen su postura y ubicación en la escena) en el *grafo de escena*.

La estructura *vaActclass* contiene información detallada de cada uno de sus grados de libertad empleados por un determinado tipo de actor, así como información pormenorizada de los puntos de articulación existentes y su organización topológica. Estos dos tipos de informaciones son almacenados en dos estructuras auxiliares relacionadas entre sí.

- La estructura ***vaDof*** que almacena toda la información referente a un grado de libertad.
- La estructura ***vaSkelprot*** que almacena toda la información referente a un punto de articulación.

Si bien la información sobre los grados de libertad y los puntos de articulación se almacenan en estructuras separadas, existe cierta relación entre ellas puesto que un punto de articulación estará formado por uno o varios grados de libertad. Esta relación y la forma en la que ambos tipos de estructuras auxiliares son almacenados dentro de la estructura *vaActclass* es mostrada en la figura siguiente:

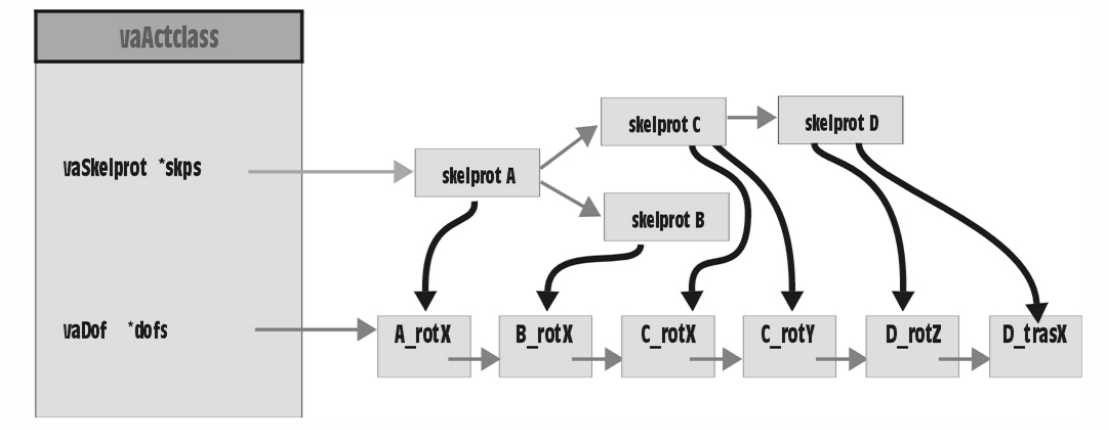


Figura 5-2. Visión general de elementos constituyentes de una *Clase de Actor*.

Como se puede observar en la *Figura 5-2*. la estructura va*Actclass* contiene una ***Lista de Grados de libertad (****dofs****)*** y una ***Lista de Puntos de Articulación (****skps****).*** Las estructuras de tipo *vaDof* se almacenan en una lista encadenada, mientras que las estructuras *vaSkelprot* se organizan de una forma jerárquica que es empleada para describir la topología del actor. También se puede observar como cada punto de articulación contiene referencias a los grados de libertad que utiliza.

En los siguientes apartados se va primero a definir cual es el contenido de las estructuras de datos auxiliares *vaDof y vaSkelprot*, presentándose a continuación la estructura de datos *vaActclass*, y las relaciones existentes entre la representación de un actor en el *Grafo de escena* y su *vaActclass*.

5.3.1 Información referente a Grados de Libertad: Estructura vaDof.

Como se había descrito en el apartado que trata sobre los nodos *Skeleton*, cada punto de articulación de un actor puede tener entre uno y cuatro grados de libertad. Desde el punto de vista del *grafo de escena* un grado de libertad es un valor numérico que define un valor angular concreto (en el caso de grados de libertad rotacionales) o un desplazamiento (en el caso de un grado de libertad traslacional). La lista de estos valores numéricos es almacenada en la tabla ***conf*** del nodo *Actor*.

La información de un grado de libertad que es necesaria para la representación de un *grafo de escena* es mínima (un simple valor numérico), sin embargo, para la realización de una gestión adecuada de un grado de libertad serían necesario un mayor número de parámetros. En la *Figura 5-3* se muestra la articulación del codo de un actor humanoide, y sobre ella aparecen representados los principales parámetros que intervienen en la definición de su único grado de libertad. Para cada grado de libertad de una determinada articulación existen unos límites físicos más allá de los cuales no se puede llegar sin que la articulación resulte dañada (estos limites aparecen etiquetados en la imagen como ***min*** y ***max***). También existe un segundo margen de movimiento más restringido que el anterior, que si bien no representan los límites físicos del recorrido hace referencia al rango de movimiento ¨no forzado¨ (los valores extremos de este margen aparecen etiquetados en la imagen como ***rMin*** y ***rMax***). Resulta, además, adecuado definir un valor concreto del grado de libertad que pueda ser considerado como posición de descanso (etiquetada como ***def*** en la figura).

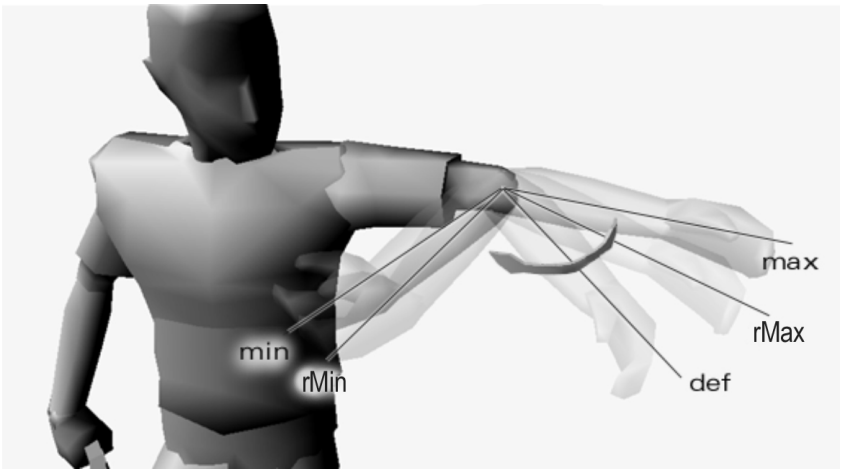


Figura 5-3. Posición de descanso, rango de trabajo no forzado y topes máximo y mínimo del grado de libertad de la articulación del codo de un actor humano.

Las estructura *vaDof* encargadasde contener una definición completa de los grados de libertad de un actor, son almacenadas dentro de una lista enlazada en cada *vaActclass*. Su contenido es el siguiente:

```
typedef struct vaDofStruct{
   char    name[VA_MAX_NAMELENGTH];                  // Nombre de este grado de libertad .
   int     dofIndex;                                 // Indice del grado del libertad.
   float   min, max;                                 // Valores extremos minimo y maximo.
   float   def;                                      // Valor por defecto (posicion de descanso).
   float   rMin, rMax;                               // Rango de trabajo no forzado.

   int     sklIndex;                                 // Indice del esqueleto al que pertenece.
   void *  extensionSlots[VA_MAX_EXTENSION_SLOTS]; //soporte para extensiones.

   vaDof * next;                                     // Puntero al siguiente dof en la lista.
}vaDof;
```

En esta estructura se almacenan las cotas mínima y máxima del grado de libertad campos ***min*** y ***max***), también el rango de trabajo no forzado (campos ***rMin*** y ***rMax***), y la posición por defecto (campo ***def***). También se observan dos índices, el primero de ellos (***dofIndex***) es un entero que actúa como identificador dicho grado de libertad dentro del actor, y el segundo (***sklIndex)***, almacena el identificador del punto de articulación al que pertenece. El campo ***name*** almacena el nombre con el cual se reconocerá a dicho grado de libertad, el campo ***extensionSlots*** permite que se puedan ampliar el número de campos existentes en esta estructura (la capacidad de extensión de esta estructura será detallada en un apartado posterior), y por último ***next*** es un puntero al siguiente elemento en la lista enlazada.

5.3.2 *Información referente a los puntos de articulación: Estructura vaSkelprot.*

En cada *Clase de Actor* se almacena información suficiente para definir la estructura esquelética de un actor, esta información es recogida en una única estructura de datos a la que se denomina ***vaSkelprot***. La estructura *vaSkelprot* almacena toda la información necesaria para definición de un punto de articulación de un actor virtual, el nodo *Skeleton* almacena el subconjunto de esta información que es imprescindible para la gestión del *grafo de escena*. La estructura *vaSkelprot* está formada por siguientes bloques de información.

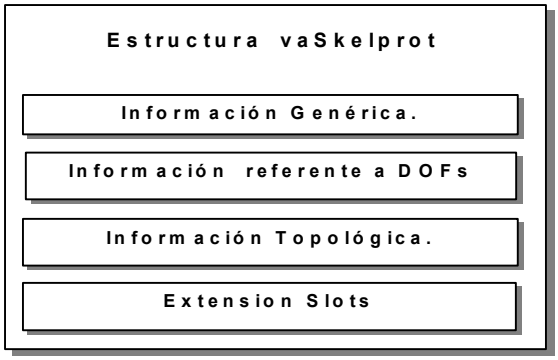


Figura 5-4. Elementos constitutivos de una estructura *vaSkelprot*.

Tal y como se muestra en la *Figura 5-4*, dentro de la estructura ***vaSkelprot*** existe un bloque de información que almacena datos de tipo general, tales como el nombre de la articulación, su identificador, o la información relativa a *offsets* respecto a su articulación padre. Otro bloque almacena la información referente a los grados de libertad que constituyen dicha articulación, otro, se encarga de almacenar las relaciones topológicas entre articulaciones, y por último, un cuarto bloque dentro de la estructura *vaSkelprot* hace referencia a su capacidad para integrar informaciones que pueden ser necesarias en ampliaciones futuras.

El contenido de la estructura *vaSkelprot* es el siguiente:

```
typedef struct vaSkelprotStruct{
   char    name[VA_MAX_NAMELENGTH];      // Nombre de la articulacion.
   int     sklIndex;                     // Indice en la lista de skeletons.
   float   offsetPos[3];                 // Offset de traslacion.
   float   offsetHpr[3];                 // Offset de rotacion. Orden ry->rz->rx.
   float   offsetMat[16];                // Matriz compuesta con offsetPos y offsetHpr.
   float   lodOffDistance;               // Distancia de desactivacion  por LOD.

   int     handlerType;
   float   handlerDist;

   vaDof * rz;                           // Grados de Libertad
   vaDof * ry;
   vaDof * rx;
   vaDof * tx;

   int         nSons;                     // Numero hijos.
   vaSkelprot * son;                     // Nodos hijos.
   vaSkelprot * fath;                    // Nodo padre.
   vaSkelprot * brPrev;                  // Hermano anterior
   vaSkelprot * brNext;                  // Hermano siguiente.

   void *  extensionSlots[VA_MAX_EXTENSION_SLOTS];
}vaSkelprot;
```

La finalidad de los distintos campos que constituyen esta estructura es detallada en los siguientes subapartados.

5.3.2.1 Información de tipo genérico.

Cada punto de articulación de una determinada *vaActclass* es identificado mediante una cadena de texto que es almacenada en el campo ***name*** de la estructura *vaSkelprot*, existen funciones que permiten acceder a la información de una articulación a través de su nombre.

Los campos ***offsetPos***, ***offsetHpr*** almacenan los valores de los *offsets de traslación* y *rotación* entre un punto de articulación y su punto de articulación padre. El significado de estos campos ha sido detallado anteriormente en el apartado referente a los nodos *Skeleton*.

El campo ***lodOffDistance*** indica la distancia a la que los movimientos de esta articulación dejan de ser

perceptibles. Es empleada para la gestión de nivel de detalle topológico que será explicada en el siguiente capítulo.

Los campos ***handlerType*** y ***handlerDist***, proporcionan información sobre la interfaz 3D de manipulación de ese punto de articulación, su finalidad ya ha sido mostrada cuando se ha explicado la estructura *vaSkeleton*.

5.3.2.2 Información referente a Grados de Libertad.

Cada punto de articulación almacena cuatro punteros a estructuras de tipo *vaDof*. Los campos ***rz***, ***ry***, ***rx*** y ***tx***, son, respectivamente punteros a los grados de libertad de rotación sobre el eje Z, el eje Y, y el eje X, y el grado de libertad de traslación a lo largo del eje X. De forma similar a como ocurría en los nodos *Skeleton,* se puede indicar que un determinado grado de libertad no existe en dicha articulación, haciendo que el contenido de dicho puntero sea *NULL*. Como se puede observar en la *Figura 5-2*, por medio de estos punteros se define la relación existente entre las estructuras *vaSkelprot* y *vaDof*.

5.3.2.3 Información referente a la Estructura Topológica.

La estructura *vaSkelprot* incorpora los campos necesarios para poder definir la dependencia jerárquica existente entre los distintos puntos de articulación (ver *Figura 5-5*). Los campos ***nsons***, ***son***, ***fath***, ***brPrev*** y ***brNext*** almacenan respectivamente el número de hijos que tiene un nodo determinado, y punteros al primero de sus hijos, al que actúa de padre, y a los puntos de articulación que actúan dentro del grafo como hermanos anterior y posterior.

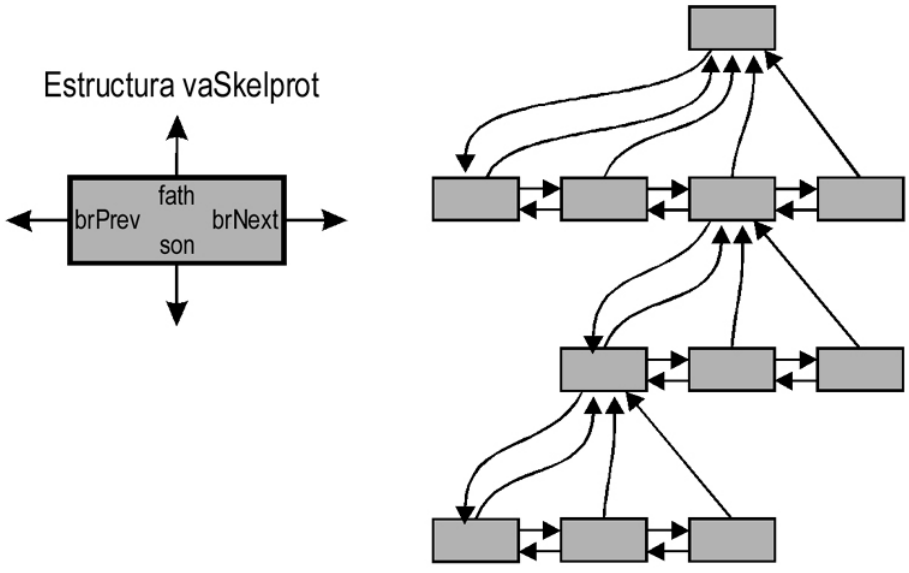


Figura 5-5. Estructura jerárquica de articulaciones definida mediante un grafo de estructuras *vaSkelprot*.

Mediante estos enlaces se consigue que la estructura topológica del actor virtual esté totalmente definida dentro de la estructura *vaActclass*, cada actor de una clase determinada replica dicha topología en la organización de sus nodos *Skeleton*.

5.3.2.4 Soporte para ampliaciones. Extension Slots.

Dentro de la estructura *vaSkelprot* se pueden añadir nuevos campos que sirvan para ampliar las funcionalidades de los actores sintéticos, así por ejemplo tendría sentido almacenar informaciones sobre masa, momentos de inercia, espacios de alcance, puntos auxiliares para facilitar los cálculos de detección de colisiones, etc. Estas informaciones son almacenadas dentro del campo ***extensionSlots***. El método para realizar este tipo de extensiones sobre los actores virtuales es explicada en detalle en el apartado 5.4.1.

5.3.3 Estructura de datos principal.

Una vez detallados las estructuras auxiliares que intervienen en la definición de una *Clase de Actor*, veamos en cual es la estructura principal que almacena su información.

```
typedef struct vaActclassStruct{
   char        name[VA_MAX_NAMELENGTH];

   int              nSkps;                  // Numero de articulaciones.
   vaSkelprot  * skps;                      // Jerarquia de estructuras vaSkelprot.
   vaSkelprot ** pSkps;                     // Tabla de punteros a vaSkelprot.

   int              nDofs;                  // Numero de grados de libertad.
   vaDof *          dofs;                   // Lista de vaDof.
   vaDof **         pDofs;                  // Tabla de punteros a vaDofs.

   int       numTopoLods;                   // Informacion sobre los niveles de detalle
   vaTopoLodInfo **topoLods;                // topologicos.

  float     sklsrootBsph[4];                // Bounding sphere del cuerpo del actor.

   void *                                   behaviourData;
   vaActorBehaDataCreateFunc        createBehaviourDataFunc;
   vaActorBehaDataDeleteFunc        deleteBehaviourDataFunc;

   vaActorBehaFunc          refpointBehaviourFunc;
   vaActorBehaFunc          sklsrootBehaviourFunc;
   vaActorBehaFunc          dofsBehaviourFunc;

   vaActorDrawFunc          preDrawFunc;
   vaActorDrawFunc          postDrawFunc;

   int    nActors;
   vaActor **lActors;             //vector de de punteros a sus actores

   int     numExtensionSlots;
   void *  extensionSlots[VA_MAX_EXTENSIONS];

   vaActclass *prev,  *next;
}vaActclass;
```

En primer lugar se observa el campo ***name***, que contiene el nombre a través del cual se identifica. Los campos ***nSkps***, ***skps*** y ***pSkps***, son empleados para almacenar la información sobre la estructura articulada de esa *clase de actor,* empleando para ello las estructuras de tipo *vaSkelprot* ya detalladas en el apartado anterior. De modo similar, los campos ***nDofs***, ***dofs*** y ***pDofs***, son empleados para almacenar la lista de grados de libertad existentes en esa *Clase de Actor*, empleando las estructuras *vaDof* cuyas características ya han sido detalladas en el apartado 5.3.1.

La información sobre detalles de nivel topológico es almacenada en los campos ***numTopoLods*** y ***topoLods***. Cada nivel de detalle está definido a partir de una estructura del tipo *vaTopoLod*, que presenta el siguiente contenido:

```
struct  vaTopoLodStruct{
        float distMin, distMax;          // Margen de distancias de aplicación de este LOD.
        float cullCostl;                 // Estimación sobre su coste de cull.
        float cpuProcessCost;            // Estimación sobre coste de procesado de sus articulaciones empleando.
        float hardgrafProcessCost;       // la cpu y el hardware gráfico.
}vaTopoLod;
```

Los niveles de detalle topológicos de un actor se generan a partir de los valores de los campos *lodOffDistance* almacenada en las estructuras *vaSkelprot*. En cada estructura *vaTopoLod* se almacenan los márgenes de distancia en los cuales actúa, y, además, unas estimaciones relativas coste de procesado del CULL y de gestión de transformaciones de los nodos *Skeleton* empleando el hardware gráfico o la CPU, estos valores serán empleados por los métodos de gestión del culling y nivel de detalle que serán explicados posteriormente.

El campo ***behaviourData*** almacena los distintos tipos de datos que el usuario pueda necesitar para definir los modelos motor y comportamental de esta clase de actor. Las funciones ***createBehaviourDataFunc*** y ***deleteBehaviourDataFunc*** se emplean para poder definir unas funciones de callback que se encargaran de rellenar el contenido del campo *behaviourData* de los nodos *Actor* pertenecientes a esta clase.

Los campos ***refpointBehaviourFunc***, ***sklsrootBehaviourFunc*** y ***dofsBehaviourFunc*** definen el contenido por defecto que tendrán los campos de mismo nombre existentes en las estructuras *vaActor*. Las funciones ***preDrawFunc*** y ***postDrawFunc*** permiten aplicar algunas modificaciones temporales sobre la forma en la que se dibujaran los actores virtuales, lo cual proporciona una forma sencilla de hacer que varios actores de la misma clase presenten una apariencia externa diferenciada.

Los campos ***nActors*** y ***lActors*** definen una lista de punteros a todos los nodos *Actor* pertenecientes a esta clase.

Los campos ***numExtensionSlots*** y ***extensionSlots*** permiten almacenar distintos tipos de informaciones de alto nivel y están pensados para almacenar la información que pueda ser necesaria en futuras ampliaciones.

Por último se pueden observar los campos ***prev*** y ***next*** que hacen que las distintas clases de una escena estén organizadas en una lista doblemente enlazada.

5.3.4 Relaciones entre la representación de un actor en el Grafo de escena y su vaActclass.

Como ya se ha citado anteriormente, la información asociada a un actor virtual se encuentra almacenada en dos zonas distintas, las informaciones de tipo particular o directamente relacionadas con su dibujado se encuentran almacenadas en los nodos *Actor* y *Skeleton* del *grafo de escena*; las informaciones de tipo común que no están directamente relacionadas con el dibujado y son comunes a todos los actores de la misma clase son almacenadas en su correspondiente estructura *vaActclass*. Esta dualidad fuerza a que exista de una estrecha relación entre la representación del actor en el *grafo de escena,* y su representación en la correspondiente *Clase de Actor*. Existen dos tipos de relaciones principales, la forma en la que se almacena la información topológica, y el método de acceso a la información de los puntos de articulación y los grados de libertad. En este apartado se tratan ambas relaciones y su utilidad.

5.3.4.1 Organización topológica.

La estructura jerárquica que relaciona los *nodos Skeleto*n de un actor en el *grafo de escena* es replicada en las estructuras *vaSkelprot* de su correspondiente *vaActclass* (ver *Figura 5-6*). Consecuentemente todos los actores pertenecientes a una determinada *Clase de Actor* presentan estructuras topológicas idénticas. Esta característica permite la existencia de métodos capaces de generar un nuevo actor virtual en el *grafo de escena* empleando solamente la información topológica almacenada en la estructura *vaActclass*.

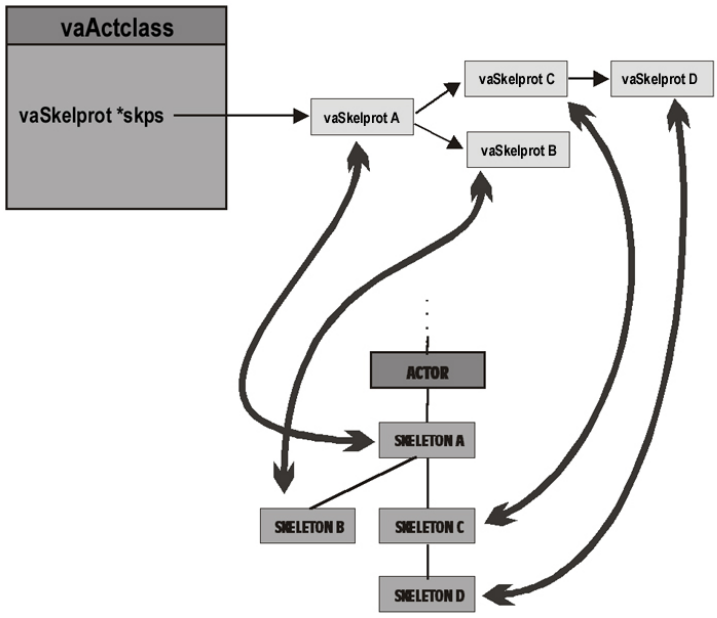


Figura 5-6. Estructura topológica del actor virtual replicada en el *Grafo de Escena* por medio de nodos *Skeleton* y en el *vaActclass* por medio de estructuras *vaSkelprot*.

5.3.4.2 Métodos de acceso a la información.

La relación más importante entre un nodo *Actor* y su correspondiente *vaActclass,* se establece a nivel del método empleado para acceder a la información de sus grados de libertad y sus puntos de articulación: Del mismo modo que el nodo *Actor* tiene una tabla de punteros que le permiten acceder a sus nodos *Skeleton*, la estructura *vaActclass* tiene una tabla de punteros (de nombre ***pSkp***) que permite acceder a sus estructuras *vaSkelprot*. Algo similar ocurre con la información referente a los grados de libertad: el nodo *Actor* presenta una tabla (de nombre ***conf***) en la que almacena los valores de los grados de libertad, y la estructura *vaActclass* presenta una tabla equivalente formada por punteros a todas las estructuras de tipo *vaDof* que intervienen en su definición (de nombre ***pDofs)***.

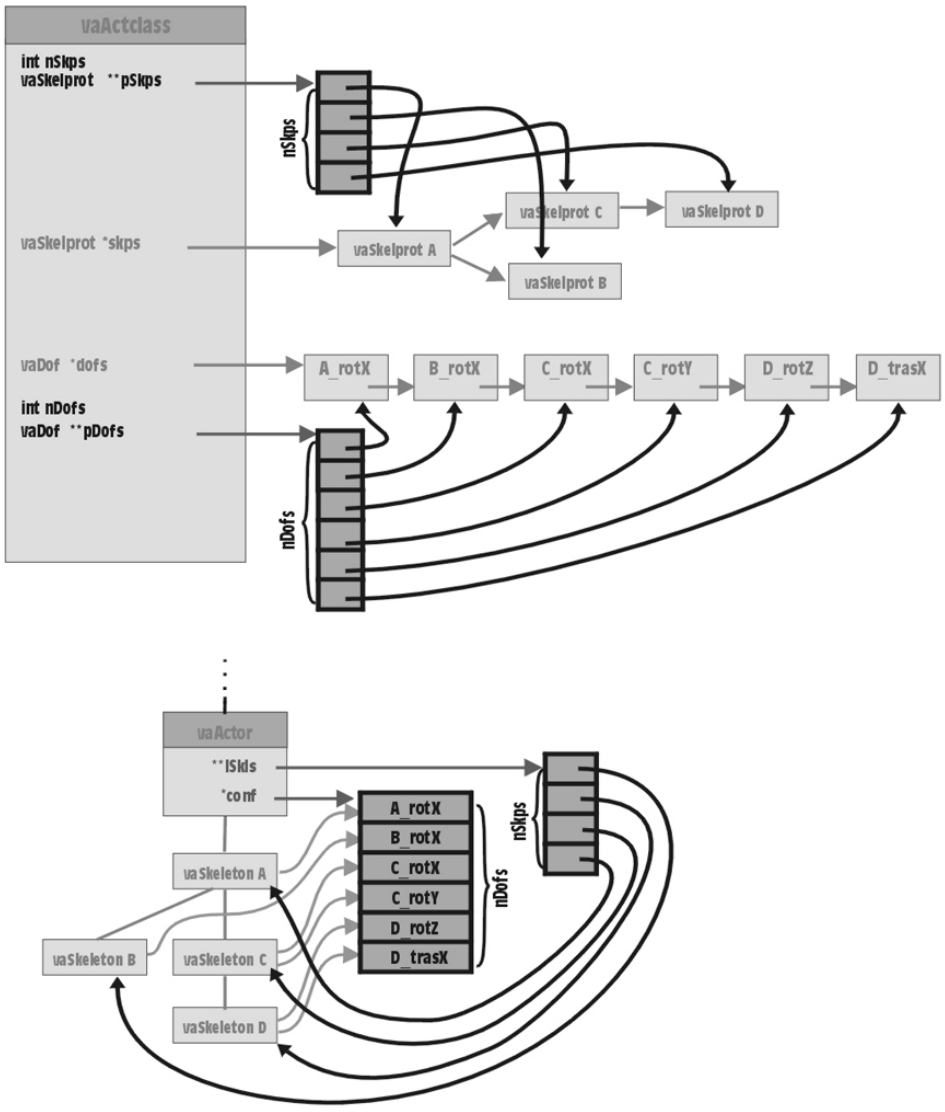


Figura 5-7: Homogeneización del acceso a la información almacenada en el *Grafo de Escena* y en las *vaActclass* mediante la utilización de vectores.

La existencia de estas tablas de punteros permite que tanto los grados de libertad como los puntos de articulación puedan ser identificados mediante un índice. El hecho de que estas listas estén replicadas

también en los nodos *Actor* del *grafo de escena,* permite que a través de dichos índices se pueda acceder a toda la información de un determinado punto de articulación o grado de libertad.

Tanto la estructura *vaDof,* como la *vaSkelprot* tienen un campo (*name*) en el cual almacenan el nombre de dicho grado de libertad o punto de articulación. Dicho nombre actúa como identificador dentro de la *Clase de Actor*, y en consecuencia cada nombre ha de ser único dentro de cada *clase de actor*. La utilización de una cadena de caracteres como identificador proporciona un método muy intuitivo acceder a un determinado punto de articulación o grado de libertad, sin embargo, la localización de una cadena dentro de todas las existentes es una operación costosa que resulta inadecuada para su utilización en tiempo real. El acceso por medio de índices no es tan intuitivo, pero resulta muy rápido, y proporciona un método de acceso unificado a la información, que resulta independiente de si ésta se encuentra en el *vaActclass* o en el nodo *Actor*. Existen métodos para localizar el índice correspondiente a un determinado punto de articulación o grado de libertad a partir de su nombre.

La existencia de estas tablas de indexación, permite que las relaciones de dependencia entre un actor y su *vaActclass* sean mínimas, permitiendo que se reduzca a una simple conexión entre el nodo *Actor* y su *vaActclass* (mediante su campo ***vaActclas***) y evitando relaciones más complejas tales como la existencia de punteros entre los nodos *Skeleton* y sus correspondientes *vaSkelprot*.

5.4 Estructura *vaActclass* como soporte para ampliaciones.

Uno de los objetivos principales de este trabajo es proporcionar un sustrato básico estable que sea capaz de absorber fácilmente futuras ampliaciones. Los nodos *Actor* y *Skeleton* proporcionan un método estable para definir y gestionar el aspecto geométrico de un actor virtual. La estructura *vaActclass* actúa como almacén de las informaciones de alto nivel, y está pensada para actuar como contenedor de todas las estructuras de datos que pueden ser necesarias en cualquier futura ampliación.

Tanto la estructura principal que almacena la información de una *Clase de Actor* (*vaActclass*), como las estructuras auxiliares que almacenan la información de sus grados de libertad y puntos de articulación (*vaDof* y *vaSkelprot)* están ideadas de tal modo que son capaces de almacenar campos que permitan ampliar las capacidades de los actores virtuales. El nodo *Actor* también dispone de esta misma capacidad. En la *Figura 5-8* se puede observar cuales son las zonas en las que un futuro desarrollador pueda introducir los campos que necesite para la implementación de alguna extensión.

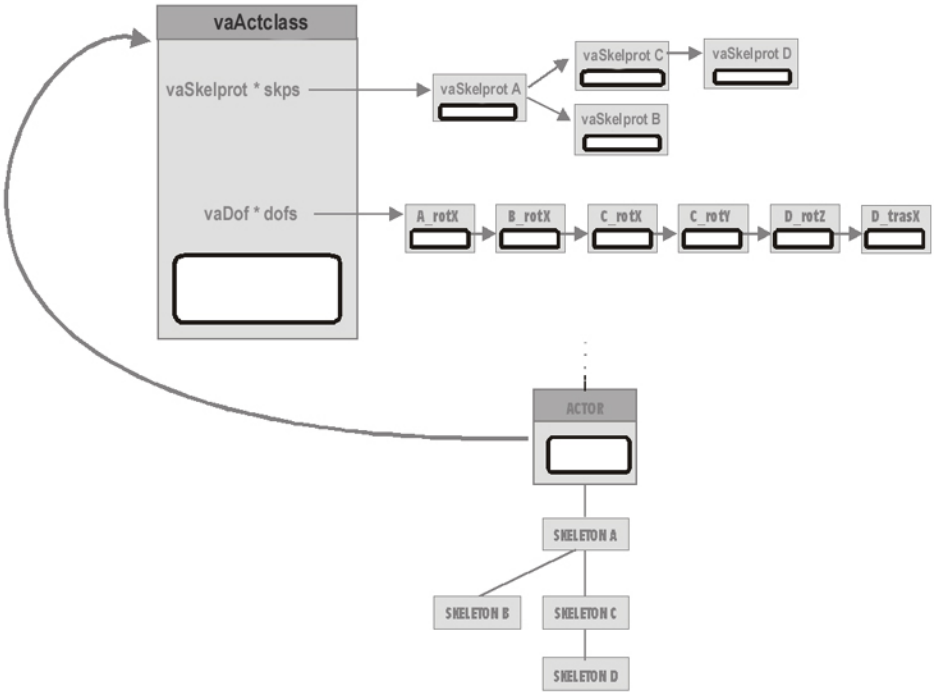


Figura 5-8: Zonas preparadas para la ampliación de las características de los actores virtuales.

Tanto la estructura *vaActclass* como las estructuras *vaSkelprot, vaDof* y el nodo *Actor* presentan un campo de nombre ***extensionSlots*** que está pensado para almacenar las informaciones que sean necesarias para la implementación de una determinada extensión. Este campo consiste en todos los casos en una tabla indexada, de tal modo que todas las informaciones asociadas a una determinada extensión emplean un mismo índice. Así, si una determinada extensión está almacenada en el slot 5 de la tabla *extensionSlots* del *vaActclass*, la informaciones relativas a los puntos de articulación estarán almacenadas en los slots 5 de los *extensionSlots* de las estructuras *vaSkelprot*, la informaciones relativas a los grados de libertad en

los slots 5 de los *extensionSlots* de las estructuras *vaDof,* y la información sobre la utilización que un actor concreto hace de esta extensión se almacenará en el slot 5 del *extensionSlots* del nodo *Actor*.

La estructura de datos principal que almacena y gestiona una extensión es ***vaExtension***, las informaciones almacenadas en los *extensionSlots* de las estructuras *vaSkelprot*, *vaDof* y el nodo *Actor* son informaciones auxiliares que son definidas desde esta estructura principal. Esta estructura de datos no está asociada a ninguna *Clase de Actor* en particular, lo cual se posibilita que una misma extensión (por ejempo una gestión de Keyframing, o un módulo encargado del control del la mirada) sea empleada por diferentes *Clases de Actores*.

El contenido de una estructura *vaExtension* es el siguiente:

```
typedef void   (*vaExtRuntimeFunc)(  vaActor *act);
typedef void * (*vaExtDataCreateFunc)(void *data);
typedef void   (*vaExtDataDeleteFunc)(void *data);

typedef struct vaExtensionStruct{
   char  name[VA_MAX_NAMELENGTH];
   int   index;

   vaExtDataCreateFunc      actclassCreateFunc;
   vaExtDataCreateFunc      dofCreateFunc;
   vaExtDataCreateFunc      skpCreateFunc;
   vaExtDataCreateFunc      actorCreateFunc;

   vaExtDataDeleteFunc      actclassDeleteFunc;
   vaExtDataDeleteFunc      dofDeleteFunc;
   vaExtDataDeleteFunc      skpDeleteFunc;
   vaExtDataDeleteFunc      actorDeleteFunc;

   vaExtRuntimeFunc         actorUpdateFunc;
}vaExtension;
```

Como se puede observar, la estructura *vaExtension*, está básicamente formada por las funciones que permiten crear y borrar los distintos bloques de información. En esta estructura se almacena, además, el nombre de la extensión (***name***), el índice que actúa como identificador de la extensión (***index***) y, el puntero a la función ***actorUpdateFunc*** encargada de actualizar el estado de un actor virtual en función de la información proporcionada por cada extensión.

Cuando un actor virtual va a ser dibujado, su estado interno (básicamente sus grados de libertad), es fijado mediante la ejecución de la siguiente secuencia de código:

```
(*act->dofsBehaviourFunc)( act, act->behaviourDdata );
vaActclass *actclass = act->actclass;
for (int i = 0;  i < actclass->numExtensionSlots; i++){
        vaExtension *extslot = actclass->extensionSlots[i];
        ( * extslot->actorUpdateFunc)(act );
}
```

La primera orden es la llamada la función principal de gestión del comportamiento del actor, en ella se emplean los datos almacenadas el campo *behaviourData* del nodo *Actor* para actuar directamente sobre los grados de libertad del actor, o bien actuar sobre otros parámetros que actuarán sobre las extensiones. A continuación se recorren todas las extensiones del actor, invocando a la función *actorUpdateFunc* de cada una de ellas. La función *actorUpdateFunc* emplea los datos almacenados en las 4 zonas para actualizar el estado interno del actor y actuar sobre sus grados de libertad. Puede haber extensiones de alto nivel que no actúen directamente sobre los grados de libertad, así, por ejemplo puede existir una extensión que se dedique a actualizar distintos parámetros relacionados con el estado de animo de un actor, y si bien, esta extensión no actúa directamente sobre sus grados de libertad, sus parámetros puede ser aprovechados por otras extensiones que determinan las expresiones faciales del actor o su forma de movimiento.

Esta filosofía ya ha sido utilizada para incorporar algunas ampliaciones tales como listas de posturas, tablas de *Keyframing* o cinemática Inversa. La forma en la que estas características han sido añadidas sobre la estructura básica de los actores virtuales es detallada en el apartado siguiente.

5.4.1 Ejemplos de integración de contenidos de Alto Nivel.

Las estructuras propuestas en esta tesis, incorporan determinadas zonas en las que un desarrollador de aplicaciones puede incorporar los campos que necesite para integrar un determinado mecanismo de gestión de alto nivel. El objetivo de este apartado es describir la forma en la que informaciones dispares pueden ser integradas empleando los mecanismos de extensión propuestos en el apartado anterior, y mostrar algunos ejemplos de este tipo extensiones. Este apartado se centra en los aspectos relativos a la integración, sin entrar a realizar descripciones detalladas del funcionamiento interno de las estructuras de datos empleadas.

Cuando se define una extensión se ha de determinar que bloque de información ha de ser almacenada en cada una de las cuatro zonas de extensión. Una forma adecuada sería definir una estructura de datos para cada bloque de información, así, por ejemplo, si se estuviese implementado una extensión de nombre "*sample*", la estructura que almacena la información de los grados de libertad podría llamarse *sampleDofData*, la de los puntos de articulación *sampleSkpData*, la del *vaActclass* sería *sampleActclassData*, y la del nodo *Actor* sería *sampleActorData*. En algunos casos, puede ser necesario que estas estructuras empleen de forma interna otras estructuras auxiliares. En los ejemplos que se mostrarán a continuación, primero se presentan las estructuras de datos principales que serán almacenadas en los *extension slots*, a continuación algunas estructuras auxiliares, y por último se explica como está implementada la función de *actorUpdateFunc* encargada de actualizar el estado interno del nodo *Actor* en función del contenido de dicha extensión. La información sobre cada extensión mostrada aquí es muy abreviada, para observar todos los detalles de la implementación de una extensión se puede consultar el capítulo final en el que se muestra de una forma detallada la implementación de un ejemplo de aplicación.

5.4.1.1 Posturas.

Una postura es una conjunto de valores concretos de grados de libertad que colocan a un actor virtual en una determinada una actitud (acostado, sentado etc.). Una postura está definida por un conjunto de parejas formadas por un índice de grado de libertad y el valor que se desea que adopte. Es habitual que una postura contenga valores para todos los grados de libertad del actor. También es posible definir una postura que defina un conjunto reducido de valores que modifiquen una determinada zona del cuerpo, así, es posible tener estructuras de tipo postura, que sólo almacenen información referente a una determinada expresión facial, o a una determinada posición de las manos.

Para definir una extensión de gestión de posturas es suficiente con almacenar datos en dos de las zonas de expansión: En los *extensionSlots* del *vaActclass* es almacenada la estructura *poseActclassData* y la estructura *poseActorData*, es almacenada en el nodo *Actor*:

```
typedef struct poseActclassDataStr{
   int          numPoses;       // Numero de posturas predefinidas.
   vaxPose   *poses[];          // Lista de posturas predefinidas.
}poseActclassData;

typedef struct poseActorDataStr{
   int              numCurPoses;  // Numero de posturas en ejecucion.
   vaxActPose  *curPoses;         // Lista de posturas en ejecucion.
}poseActorData;
```

Estas estructuras emplean a su vez las estructuras auxiliares *vaxPose*, encargada de almacenar la información de una postura, y *vaxActorPose*, encargada de almacenar el estado de aplicación de dicha postura en un actor virtual concreto.

```
ttypedef struct vaxPoseStr{
   int         numDofs;
   int         dofIndexes[];       // Indices de los dofs a los afecta
   float       dofValues[];        // Valores que fija.
}vaxPose;

typedef struct vaxActPoseStr{
   int          poseIndex;       // Indice de la pose.
   float        startTime;       // Instante de inicio de la transicion.
   float        transitionTime;  // Duracion de la transicion.
   struct vaxActPoseStr *next;
}vaxActPose;
```

La función de *actorUpdateFunc* encargada de actualizar el estado interno del nodo *Actor* en función del contenido de su extensión de posturas tiene el siguiente aspecto:

```
static void poseActorRuntimeFunc(vaActor *act){
   int slotIndex          = poseGetExtIdentifier();   // Obtiene el indice del extension slot  en el que se encuentra.
  poseActorData     *actorData          = (poseActorData *)     act->extensionSlots[ slotIndex];
  poseActclassData *actclassData      = (poseActclassData *) act->actclass->extensionSlots[ slotIndex];

   float curTime      = utilityGetCurTime();                    // Tiempo local de la simulacion.
   vaxActPose *actPose = actorData->curPoses;
   while (actPose != NULL){
       vaxPose *pose = actclassData->poses[actPose->poseIndex];
       float  value  = (curTime - actPose->startTime)/actPose->transitionTime;
       if (value <= 1.0f)   poseAppyToActor(act, pose, value);    //Hace que el actor adopte esta postura
       else                 poseRemoveFromActor(act, actPose);//Quita la postura de la lista del actor.
       actPose = actPose->next;
   }
}
```

La función *utilityGetCurTime()* retorna el tiempo desde que comenzó la simulación, este valor es empleado para obtener un valor normalizado que será empleado por la función *poseAppyToActor()* encargada de fijar los valores correspondientes sobre los grados de libertad del actor virtual, la función *poseRemoveFromActor()* elimina una postura dela lista de posturas del actor. La función *poseGetExtIdentifier(),* retorna el índice de los *extension slots* en los que se almacena la información referente a posturas.

5.4.1.2 Keyframing.

Una determinada secuencia de movimiento de un actor puede ser almacenada en forma de la secuencia de valores que toma sus grados de libertad en sucesivos instantes de tiempo. El movimiento puede ser especificado indicando los valores existentes en cada frame, o bien seleccionando unas cuantos instantes clave de dicho movimiento empleando algún método de interpolación para generar los valores intermedios (keyframing).

Para definir una extensión de gestión de keyframing se ha de definir una estructura que es almacenada en el *extensionSlots* de *vaActclass* (*kftableActclassData*), y otra que es almacenada en los *extensionSlots* del nodo *Actor* (*kftableActorData*):

```
typedef struct kftableActclassDataStr{
   int                numKftables;        // Numero de tablas de keyframing.
   vaxKftable      * kftables[];          // Lista de tablas de keyframing.
}kftableActclassData;

typedef struct kftableActorDataStr{
   int                numCurKftables;    //Numero de tablas de keyframing en ejecucion.
   vaxActKftable   * curKftables;        //Lista de tablas de keyframing en ejecucion.
}kftableActorData;
```

Se puede observar como estas estructuras emplean a su vez las estructuras auxiliares *vaxKftable*, encargada de almacenar una secuencia de movimiento, y *vaxActKftable* que almacena información sobre el estado de aplicación de una tabla de keyframing sobre un actor virtual concreto.

La estructura *vaxKftable* almacena una tabla con los valores de los grados de libertad de un actor en distintos instantes, bien sean los valores correspondientes a una secuencia continua de frames o bien frames clave (keyframes). Al igual que ocurría con las posturas, una estructura de este tipo puede almacenar información sobre todos los grados de libertad del actor, o bien tan sólo el movimiento de una parte del cuerpo, esto permite que por ejemplo en un actor humano se puedan almacenar secuencias de movimientos de las manos o de movimientos faciales.

```
typedef struct vaxKftableStr{
   int     numDofs;
   int    *dofIndexes;          // Indices de los dofs a los afecta
   int     numKfTimes;          // Numero de KeyFrames.
   float  *kfTimes;             // Lista de instante temporal asociado a cada keyframe.
   float  *dofValues;           // Tabla de 2 dimensiones con los valores
   float   totalTime;           // Duracion total de la tabla.
}vaxKftable;

typedef struct vaxActKftableStr{
   int    kftIndex;             // Indice de la kftable en la lista del actclass.
   float  startTime;            // Instante de inicio de ejecucion de la tabla.
   struct vaxActKftableStr *next; //puntero al siguiente.
}vaxActKftable;
```

La función de *actorUpdateFunc* encargada de actualizar el estado interno del nodo *Actor* en función del contenido de su extensión de keyframing tiene el siguiente aspecto:

```
static void kftableActorRuntimeFunc(vaActor *act){
   int slotIndex       = kftableGetExtIdentifier();    // Indice del extension Slot.
   kftableActorData     *actorData       = (kftableActorData *)act->extensionSlots[slotIndex];
   kftableActclassData *actclassData   = (kftableActclassData *)act->actclass->extensionSlots[slotIndex];
   float curTime      =  utilityGetCurTime();                    // Tiempo local de la simulacion.
   vaxActKftable *actKftable = actorData->curKftables;
   while (actKftable != NULL){
      vaxKftable *kftable = actclassData->kftables[ actKftable->kftIndex ];
      float localTime = curTime - actKftable->startTime;
      kftableApplyToActor(act, kftable, localTime);  //Aplica los valores al actor, realiza interpolacion si es necesario.
      actKftable = actKftable->next;
   }
```

La función *kftableApplyToActor()* modifica el valor de los grados de libertad necesarios a partir del contendido de la tabla de keyframing, y del valor de *localTime* que indica la posición dentro de la tabla en la que estamos, realizando la interpolación entre los valores de los keyframes.

5.4.1.3 Cinemática Inversa.

El elemento básico para la implementación de la cinemática inversa es la cadena cinemática, la cual consiste en una lista de puntos de articulación que unen un determinado punto de articulación que

permanece estático (al que suele denominar "*root"*), con otro que está sufriendo una modificación (al que se denomina "*effector"*). El código de gestión de la cinemática inversa ha de calcular automáticamente, los valores que han de adoptar los grados de libertad de los puntos de articulación intermedios.

La colocación de un "*effector"* en una determinada posición puede ser realizada de múltiples formas, así por ejemplo la colocación de la mano de un actor humanoide en una determinada posición puede ser llevada a cabo implicando el movimiento del codo, del codo y el hombro, o incluso del codo, el hombro y la columna vertebral. También se pueden establecer restricciones en el movimiento que fuercen a que la orientación de la mano sea siempre horizontal (como sí llevase una bandeja), o vertical (como sí empujase un objeto). También resulta adecuado poder realizar distintos tipos de ajustes sobre la precisión del resultado y consecuentemente sobre la rapidez con la que se realizan los cálculos implicados. Un mismo "*effector"* puede disponer de varias cadenas cinemáticas y es adecuado poder seleccionar la que se desea aplicar en cada momento.

Para la implementación de una extensión de soporte para cinemática inversa es conveniente almacenar datos en las cuatro zonas de expansión, así, se van a definir 4 diferentes estructuras de datos, que serán almacenadas en los *extensionSlots* del *vaActclass* (*ikActclasData*), de los grados de libertad (*ikDofData*), de los puntos de articulación (*ikSkpData*) y del nodo *Actor* (*ikActorData*). Dichas estructuras tienen el siguiente contenido:

```
typedef struct ikActclassDataStr{
        int             numIkchains;        // Numero de cadenas.
        ikChain *       ikChains[ ];        // Lista total de cadenas.
}ikActclassData;

typedef struct ikDofDataStr{
        float           dofWeight;          //Resistencia al movimiento de este grado de libertad.
}ikDofData;

typedef struct ikSkpData{
        float           sklWeight;          // Resistencia al movimiento de este punto de articulación.
        int             numIkChains;        // Número de cadenas cinemáticas que tienen este punto de
                                            // articulación como  effector.
        int             ikChains[ ];        // Lista de indices de dichas cadenas.
        int             byDeffectIkChain;   // Indice de la cadena empleada por defecto.
}ikSkpData;
typedef struct ikActorDataStr{
        int                     numCurIkChains; // Número de cadenas cinemáticas en ejecución.
        actorIkChain            curIkChains[ ];     // Lista de las cadena cinemáticas en ejecución.
}ikActorData;
```

En el campo *ikChains* de la estructura *ikSkpData* se almacena una lista con todas las cadenas cinemáticas que pueden emplear este punto de articulación como *"effector"*, almacenándose en el campo *byDeffectIkChain* el índice de la que será empleada por defecto.

Como la solución a una cadena cinemática no es única, se incorporan una serie pesos asignados tanto a los grados de libertad (campo ***dofWeight*** de la estructura *ikDofData*), como a los puntos de articulación

(campo ***sklWeight*** de la estructura *ikSkpData*) que se utilizan como criterio para seleccionar la solución más adecuada.

Se puede observar como las estructuras *ikActclassData* e *ikActorData* emplean estructuras auxiliares *ikChain*, que almacena la definición de una cadena cinemática, y *actorIkChain*, que almacena información sobre el estado de aplicación de una cadena cinemática en un actor concreto. El contenido de dichas estructuras es el siguiente:

```
typdef  struct ikChainStr{
        int             chainRoot;          // Indice de punto de articulacion que actua como root.
        int             effector;           // Indice del punto de articulacion que actua como effector.
        int             numSkps;            // Numero de ptos de Articulacion que definen la cadena cinematica.
        int             skpsList[ ];        // Lista de ptos de articulacion que definen la cadena cinematica.
        int             restrictions;       // Mascara de bits con diferentes tipos de restricciones.
        float3          orientationVec;     // Vector de orientacion del effector.
        int             maxIterations;      // Numero maximo permitido de iteraciones.
        float           resolution;         // Precisión deseada en el resultado.
}ikChain;

 typedef struct  actorIkChain{
        int             ikChainIndex;       // Indice de la ikChain que se esta empleando.
        float           effectorPosition[3]; // Posicion de su effector.
 }actorIkChain;
```

Los campos *restrictions*, *orientationVec*, *maxIterations* y *resolution* de la estructura *ikChain*, son empleados para ajustar la forma en la que se calcula la solución a una determinada cadena cinemática.

La función de *actorUpdateFunc* encargada de actualizar el estado interno del nodo *Actor* en función del contenido de su extensión de cinemática inversa tiene el siguiente aspecto:

```
void ikActorRuntimeFunc(vaActor *act){
   int slotIndex          =  ikGetExtIdentifier();       // Obtiene el indice del extension slot  en el que se encuentra.
   ikActorData       *actorData            = (ikActorData*)      act->extensionSlots[slotIndex];
   ikActclassData    *actclassData         = (ikActclassData *)act->actclass->extensionSlots[slotIndex];
   for (int i = 0;  i < actorData->numCurIkChains; i++){
        actorIkChain *actorIkChain = actorData->curIkChains[i];
        int ikChainIndex  =  actorIkChain->ikChainIndex;
        ikChain * chain    =  actclassData->ikChains[ ikChainIndex ];
        ikApplyToActor( act,  chain,  actorIkchain->effectorPosition);
   }
}
```

La función *ikApplyToActor()* modifica los valores de los grados de libertad afectados por una cadena cinemática a partir de la posición indicada para el *effector*.

5.5 Estructura *vaActclass* como soporte para Simulación Macroscópica.

Uno de los requisitos para realizar una adecuada integración de los actores virtuales en una simulación es hacer que sean compatibles con aquellas aplicaciones que, por su complejidad, necesitan manejar informaciones de tipo macroscópico**[FER98]**. Éste podría ser el caso del simulador del trafico de una ciudad, de una aplicación que simulase el comportamiento de un museo, o de una aplicación de chat3D en la que existiesen muchos usuarios distribuidos en diferentes salas.

Para comprender la necesidad de este tipo de soporte, tomemos como ejemplo el de una aplicación que permita realizar una visita a un museo virtual, en cada una de las salas podría haber entre 10 y 30 personas, sin embargo, la cantidad total de personas existentes en el edificio del museo puede ser de varios millares. Realizar directamente una simulación de esta magnitud puede resultar inabordable, sin embargo, si se parte de que el usuario del sistema tan sólo puede estar en una sala en cada instante, bastaría con procesar y dibujar tan sólo las 30 personas existentes en esa sala en concreto, y disponer de métodos para crear nuevos actores virtuales que rellenen otras salas en el momento que el visitante decida cambiar de sala, así como métodos para eliminar a los existentes en la sala abandonada. El sistema manejaría parámetros macroscópicos de cada una de las salas del museo, tales como el número promedio de personas observando objetos, el número promedio de personas transitando, o incluso clasificaciones por edad o apariencia física. Esta información macroscópica, sería empleada para generar actores virtuales concretos en el momento en el que el usuario esté a punto de entrar en una determinada sala. El empleo de esta técnica, permite que el número de actores presente en el *grafo de escena* se mantenga siempre bajo unos límites razonables, y por otro lado, facilita enormemente la gestión de la aplicación.

La necesidad de crear y destruir actores en tiempo real presenta dos requisitos, el primero es que se necesita la existencia de alguna estructura que almacene la información y los métodos necesarios para generar de forma automática un nuevo actor, y la segunda, que cantidad de información a manejar en estas operaciones de creación y eliminación sea mínima: El vaActclass actúa como estructura de soporte para estas operaciones, y los nodos *Actor* y *Skeleton* permiten definir el fragmento de *grafo de escena* necesario para un actor empleando una cantidad de información mínima.

5.5.1 Proceso básico de creación y eliminación en tiempo real de actores virtuales.

Una *vaActclass* puede generar un nuevo actor siguiendo distintos procedimientos. En este apartado se explica cual sería el método más sencillo de generación automática de nuevos actores virtuales. Sobre esta versión inicial es posible aplicar sucesivos refinamientos que permiten obtener métodos de generación de nuevos actores muy sofisticados.

El método más sencillo de generación de un nuevo actor sería la clonación directa de un actor prototipo. Según este método, cada *vaActclass* tiene un puntero a un nodo *Actor* con todo su subgrafo de escena, cualquier nuevo actor es creado como una copia idéntica de este actor prototipo.

Los pasos básicos en el proceso de creación automática de un nuevo actor virtual mediante la clonación directa del actor prototipo serían los siguientes:

- Creación de un nuevo nodo *Actor*.
- Asignación de nombre que identifique a este nuevo actor, dicho nombre ha de ser único en toda la escena, pudiendo ser proporcionado por el usuario o bien generado de forma automática.
- Creación de la jerarquía de nodos *Skeleton* de este nuevo actor a partir de la información existente en las estructuras *vaSkelprot* de su *vaActclass*.
- Establecimiento de los enlaces a nivel de punteros existentes entre los nodos *Actor* y sus nodos *Skeleton*.
- Recorrido de todos los nodos (tanto *Actor* como *Skeleton*) del actor prototipo y copiado de las ramas que cuelgan de cada uno de ellos sobre sus equivalentes en el actor clonado (excepto las ramas que comienzan por un nodo *Skeleton*).
- Incorporación del nuevo actor a la lista de actores de la *vaActclass*.
- Creación de los datos de usuario del actor mediante la llamada a la función *createBehaviourDataFunc* de la estructura *vaActclass*.
- Creación de la lista de extensiones del nodo Actor, mediante el recorrido de lista de *extensionSlots* de su *vaActclass* y sucesivas llamadas a la función *actclassCreateFunc*.

De modo similar, la secuencia de pasos a realizar en el proceso de eliminación de este actor virtual serían:

- Recorrido de los subgrafos del nodo *Actor* y sus nodos *Skeleton* borrando su contenido.
- Eliminación de los nodos *Skeleton* accediendo a ellos desde la lista de punteros que tiene el nodo *Actor*
- Eliminación de la información sobre extensiones. Para ello se recorre la lista de *extensionSlots* de su *vaActclass* y se realizan las correspondientes llamadas a *actorDeleteFunc*.
- Eliminación de los datos de usuario del actor mediante la llamada a la función *deleteBehaviourDataFunc* de la estructura *vaActclass*.
- Eliminación del nodo *Actor* de la lista de actores de su *vaActclass*.
- Eliminación del nodo *Actor*.

El actor puede ser eliminado en su totalidad, o bien eliminar toda su estructura de *grafo de escena* excepto su nodo *Actor.* De este modo, el nodo *Actor* almacenaría la información que define las características individuales de ese actor, con esas características el actor podría solicitar a su *vaActclass* que regenerase toda su estructura jerárquica en el caso de que fuese necesario (por ejemplo porque el usuario del sistema regresase a alguna de las salas y fuese importante que las personas que se encontrase fuesen las mismas que antes, y tuviesen unas actitudes similares).

Se puede observar que tanto las estructuras de datos que se manejan como las operaciones que se realizan durante los procesos de creación o eliminación de actores, son mínimas. Esto permite que sea posible realizar múltiples creaciones y eliminaciones de actores en tiempos inferiores a un segundo, y, por tanto, que sea una técnica susceptible de ser aplicada de forma continua durante la simulación.

5.5.2 Métodos complejos de generación automática de actores virtuales.

El método de generación de nuevos actores por medio de clonación directa de un actor prototipo resulta muy sencilla, pero presenta como limitación básica el hecho de que todos los actores de cada *vaActclass* tienen exactamente el mismo aspecto. Sin embargo, esto puede ser fácilmente solucionado de varios modos, empleando distintos tipos de refinamiento sobre este método básico:

- Utilización de múltiples actores prototipo. En lugar de haber un único actor prototipo, el *vaActclass* tiene un pequeño conjunto de distintos actores, cuando se genera un nuevo actor se ha indicar el actor prototipo en concreto a partir del cual se realizara la clonación.
- Empleo de un único actor prototipo con múltiples texturas alternativas. Existe un único actor prototipo pero con distintas versiones de las texturas que pueden recubrirlo. Cuando se genera un actor se selecciona cual es la textura en concreto que se desea que tenga. Es una técnica muy sencilla, pero que presenta resultados muy interesantes. La *Figura 5-9* muestra un ejemplo de la aplicación de este tipo de técnica, el modelo geométrico empleado para los cuatro actores es exactamente el mismo, sin embargo, el hecho de que tengan texturas diferentes modifica totalmente su apariencia.
- Utilización de un conjunto de atributos intercambiables. El *vaActclass* tiene un conjunto de atributos que pueden ser combinados entre sí para generar distintos modelos de actor. Estos atributos pueden ser partes geométricas del actor, por ejemplo los pies, las manos, el pelo, los ojos, o también pueden ser las texturas que lo recubren, pudiéndose emplear diferentes texturas correspondientes a distintos colores de ojos, distintos texturas de piel, distintos estampados para las vestimentas, etc.
- Empleo de un modelo parametrizado. Existe un actor prototipo básico de características neutras, y un conjunto de métodos que pueden deformar su geometría adaptándolo a determinadas constituciones físicas, de edad, raza etc.

Figura 5-9. Ejemplo de creación de nuevos actores por medio de clonación y cambio de textura.

5.6 Conclusiones.

En este capítulo se ha mostrado la necesidad de una estructura independiente del grafo de escena, que almacene las informaciones de tipo geométrico, motor y comportamental que son comunes a todos los actores de una determinada especie. Dicha estructura, además, actúa como soporte para ampliaciones, y también para facilitar la realización de simulaciones macroscópicas. La estructura *"Clase de Actor"* (o *vaActclass*), presentada en este capítulo es el complemento necesario a los nodos *Actor* y *Skeleton* presentados en el capítulo anterior: Los nodos *Skeleton* y *Actor* proporcionan una método rígido, sencillo, estable, eficiente para integrar un actor virtual en un grafo de escena; La estructura *"Clase de Actor"* proprociona una estructura flexible, reutilizable y ampliable, capaz de almacenar todos las informaciones de alto nivel que puedan ser necesarias para la definición de un actor virtual complejo.

La estructura *vaActclass* emplea internamente dos estructuras adicionales (*vaSkelprot* y *vaDof*) que almacenan informaciónes referentes a los puntos de articulación que definen su topología, y a sus grados de libertad. Mediante la utilización de ambos tipos de estructuras se define exactamente la topologia correspondiente a una "clase de actor". Cada punto de articulación, y grado de libertad es identificado por medio de un nombre y un índice. Dentro de la estructura *vaActclass* existen sendas tablas indexadas que proporcionan un acceso rápido a estas estructuras. El hecho de que estas tablas indexadas aparezcan replicadas en los nodos *Actor*, proporciona un acceso múy rápido a cualquier la información de un determinado punto de articulación o grado de liberdad, independientemente de si esa información se encuentra almacenada en el nodo *Actor* o en su *vaActclass*, y hace que las relaciones de dependencia entre un Actor y su *vaActclass* sean mínimas (simplemente un Actor almacena un puntero hacia su *vaActclass*).

La estructura *vaActclass* está ideada, también, para actuar como contenedor de las estructuras de datos que puedan ser necesarias para cualquier futura ampliación. La extensibilidad de los actores virtuales está soportada debido a la existencia de una zonas de ampliación (denominadas *Extension Slots)* en la estructura *vaActclass*, en sus estructuras auxiliares *vaSkelprot* y *vaDof*. y tambien en el *nodo Actor.* Existe una estructura auxiliar de nombre *vaExtension* que automatiza el proceso de definición de una extensión. La estructura *vaExtension* es no depende de ninguna "*Clase de Actor"* en particular, lo cual posibilita que una misma extensión (por ejempo una extensión de gestión de Keyframing, o un módulo encargado del control del la mirada), pueda sea empleada por difentes clases de actores Para ilustrar la forma en la que actua el mecanismo de extensión, se ha mostrado de una forma genérica, la forma de integración de una extensiones de control de posturas, aplicación de tablas de keyfrming, y uso de cinemática inversa.

Por último, se ha presentado la utilidad de la estructura *vaActclass* como soporte para la realización de simulaciones de tipo macroscópico. Se han mostrar mostrado varias formas en las que la estructura *vaActclass* puede actuar como un mecanismo generador de nuevos actores virtuales, y también, como el hecho que que la información total de un actor virtual esté dividida en dos zonas (*vaActclass* y nodos del

grafo de escena), posibilita que las operaciones implicadas en la creación o eliminacíón de un actor virtual sean mínimas, y por tanto susceptibles de ser aplicadas en tiempo real.

Capítulo 6. Procesado de las matrices de transformación, métodos especiales de Culling y gestión de LOD, y multiproceso.

6.1 Índice.

6.1 Introducción.

En los anteriores apartados se ha descrito como la utilización de los nodos *Actor* y *Skeleton,* permite integrar cualquier tipo de actor virtual dentro de un grafo de escena, y como dichos nodos introducen importantes mejoras computacionales y organizativas; a otro nivel, se ha presentado la estructura *Clase de Actor* que permite almacenar todas las informaciones de alto nivel de los actores en una estructura independiente del *grafo de escena*, facilitando, además, la extensibilidad de las capacidades de los actores, y la posibilidad de realización de simulaciones macroscópicas. Sin embargo, existen distintos aspectos, relacionados con el procesado en tiempo real de una escena tridimensional, que aún no han sido analizados teniendo en cuenta las especiales características de una simulación que incorpore actores virtuales.

Como ya se ha citado anteriormente, la forma en la que se definen y procesan los *grafos de escena* actuales muestra claramente su orientación hacia escenarios estáticos, o con muy pocos sistemas de referencia distintos. Un ejemplo típico de una aplicación de este tipo sería un simulador de vuelo, en él, la mayoría de los objetos son estáticos (terreno, edificios, arboles, arboles...), y sólo existe un número muy reducido de elementos móviles (aviones y algún otro tipo de vehículo), que no son articulados, y que, además, presentan comportamientos sencillos. Los *grafos de escena* tradicionales están orientados hacia la gestión de este tipo de aplicaciones. Su atención está centrada en establecer un control sobe el número de polígonos que se envían al hardware gráfico, sin embargo, no prestan demasiada atención a los costes derivados de la existencia de múltiples sistemas de referencia, y no actúan de manera alguna sobre la gestión del comportamiento.

El caso de una escena con actores virtuales presenta unas características muy diferentes: incluye muchos sistemas de referencia (un solo actor puede tener 40 sistemas de referencia distintos, más de los que suelen ser necesarios en una aplicación de simulación tradicional completa), y la gestión del comportamiento de un actor virtual suele tener un coste computacional muy elevado.

Un *grafo de escena* que gestione múltiples actores virtuales debe, al igual que el tradicional, controlar el número de polígonos que llegan al hardware gráfico, pero, debe prestar igual o mayor atención a minimización de los cálculos derivados de la existencia de múltiples sistemas de referencia (gestión de matrices de transformación, procesado de *bounding volumes, etc.*), y a actuar sobre la gestión del comportamiento. La gestión de matrices de transformación llevada a cabo por los grafos de escena actuales resulta muy costosa, los métodos de culling y gestión de LOD tradicionales resultan insuficientes, y la gestión del multiproceso de la forma tradicional basada en una pipeline APP->CULL->DRAW resulta inadecuada. En este capítulo se replantean todos estos aspectos buscando los siguientes objetivos:

- Una *gestión óptima de todas las operaciones con matrices de transformación*, puesto que una operación que es muy costosa que se emplea de forma exhaustiva en el procesado de los actores virtuales.
- Un *método de culling especial* que minimice el número de operaciones con matrices de transformación, así como las operaciones de recálculo e intersección con las *bounding spheres* de los actores virtuales, y que, además, actúe sobre la gestión del comportamiento.
- Unos *métodos de gestión de LOD especiales* que minimicen el número de operaciones con matrices de transformación, y las operaciones de cálculo de distancias, y que actúe sobre el coste de la gestión del comportamiento de los actores.
- Un método que indique como se debería de distribuir el trabajo con actores virtuales entre las distintas CPUs en el caso de disponer de un sistema multiprocesador.

6.2 Procesado especial de las matrices de transformación.

La misión principal de los nodos *Actor* y *Skeleton* dentro del grafo de escena consiste en la generación de determinadas matrices de transformación. El proceso de recorrido del grafo de escena impone, además, que la matriz proporcionada estos nodos, tenga que ser multiplicada por la matriz existente en la cima de la pila de matrices. Éstas operaciones son realizadas de forma continua, y su elevado coste computacional supone un cuello de botella. Este problema ya había sido mostrado en el apartado que trata de los *nodos Skeleton,* y se había propuesto entre otras estrategias el empleo del hardware de multiplicación de matrices. En este apartado, se realiza un análisis más profundo sobre la necesidad de optimizar este tipo de cálculo, se define el método mediante el cual la utilización del hardware gráfico puede acelerar estas operaciones, se analiza la forma en la que actúan los grafos de escena tradicionales en relación con esta operación, y, por último, se presentan conclusiones sobre cual sería la mejor forma para gestionar este tipo de operaciones dentro de una aplicación que incorpore múltiples actores virtuales.

6.2.1 Estimación del cuello de botella existente en el procesado de las matrices de transformación.

La multiplicación de dos matrices de 4x4 es una operación costosa que requiere realizar 64 multiplicaciones y 48 sumas. El procesado de cada nodo *Actor* o *Skeleton* implica al menos la realización de una de estas multiplicaciones. Es fácil prever que en una escena en la que existen muchos actores virtuales, el tiempo consumido en este tipo de operaciones puede llegar a ser muy elevado,

Analicemos este problema a través de un ejemplo concreto. Supongamos que se está desarrollando una aplicación de simulación que necesita ser representada a una frecuencia de 50 imágenes por segundo, y en la que existen varios actores virtuales, cada uno de ellos formado por 40 *nodos Skeleton*. Si dicha simulación se ejecutase sobre un ordenador Onyx2 IR con una CPU R10000 a 195 Mhz. En una máquina de este tipo, la realización de multiplicación de 2 matrices de 4x4 en la CPU consume aproximadamente unos 1.8 microsegundos[•]. Según esto, la multiplicación de las matrices de los nodos *Actor* y *Skeleton* de cada uno de los actores virtuales consumiría un tiempo de 73.8 microsegundos (41 nodos x 1.8). En estas circustancias no se podría nunca sobrepasar la cifra de 271 actores dibujándose simultáneamente (20 milisegs por imagen / 0.074 milisegs por actor). En la práctica, esta cota sería mucho menor, puesto que la CPU en la realidad también ha de encargarse de otras muchas actividades (por ejemplo gestionar otros aspectos del recorrido del grafo de escena, o gestionar el comportamiento de los actores).

[•] La operación de multiplicación de las matrices de tal modo que la matriz absoluta sea conocida en la CPU ha de hacerse del siguiente modo:

cur_abs_matrix = cpuMultipyMatrixs(curMatrix, mat)

glLoadMatrix(curAbsMatrix)

Conteniendo la función *cpuMultiplyMatrixs()* el código mínimo para realizar la multiplicación de las dos matrices y transfiriendo la matriz resultado al hardware gráfico mediante la operación *glLoadMatrix()* de la OpenGl.

Esta limitación puede ser reducida considerablemente si se emplea de forma adecuada la capacidad de ciertas tarjetas gráficas para realizar multiplicaciones de matrices 4x4. El principal inconveniente de la utilización de dicha capacidad es que el resultado de dichas multiplicaciones permanece en el hardware gráfico, y los métodos actuales de gestión de los grafos de escena necesitan tener acceso a estas matrices.

6.2.2 Situación del hardware gráfico actual respecto a la multiplicación de matrices.

Las librerías gráficas de bajo nivel, como la OpenGl, contemplan en su conjunto de instrucciones, ordenes que operan directamente sobre matrices de 4x4 - *glMultMatrix()*, *glRotate()*, *glTranslate()*, *glScale()*, *glLoadMatrix()*, *glPushMatrix()*, *glPopMatrix()*... -. Una tarjeta gráfica con una implementación completa de la OpenGl ha de una tener memoria interna que almacene una pila de este tipo de matrices, así como métodos rápidos para realizar dichas operaciones.

A pesar de que la multiplicación de matrices forma parte de la especificación de la OpenGl, muchas de las tarjetas gráficas actuales no cumplen con este requisito. En realidad, es habitual que muchas tarjetas gráficas de gama media no tengan implementado su conjunto completo de instrucciones. Esto de debe principalmente a dos factores, por un lado las operaciones con vértices 3D y matrices, pueden ser realizadas a una velocidad relativamente aceptable empleando las extensiones multimedia de algunas CPUs (MMX, 3DNOW, etc.), y por otro, la mayoría de las aplicaciones actuales, manejan un número reducido de matrices de transformación.

En la actualidad, la multiplicación de matrices de 4x4 solamente se encuentra implementada en el hardware gráfico de más alta gama. Para realizar una estimación de la mejora en rapidez introducida por la utilización del hardware de multiplicación de matrices se ha hecho un test en una máquina Onyx2 IR con una CPU R10000 a 195 Mhz, del cual se ha obtenido que la multiplicación de matrices puede ser realizada en el hardware gráfico unas 12 más rápido que en la CPU[1]. Dada la facilidad con la que se puede paralelizar este tipo de operación, se puede aventurar que aún sería posible conseguir mejoras respecto a este valor en el caso de que fuese necesario.

Las capacidades de las tarjetas gráficas de gama media están mejorando rápidamente, siendo ya norma común que realicen operaciones de iluminación y transformación de vértices, e incorporan la capacidad de realizar multiplicaciones vectores por matrices de 4x4. Sin embargo, aún no disponen de la capacidad de multiplicaciones entre matrices 4x4. El rápido avance de este tipo de tecnología permite suponer que en un periodo de tiempo relativamente corto las tarjetas de gran consumo incorporarán implementaciones completas de OpenGl que incluyan este tipo de operación.

---

[1] La multiplicación de una matriz en el hardware gráfico se realiza mediante una simple llamada a:

glMultmatrixf(mat)

Mediante esta orden la matriz *mat* es transferida al hardware gráfico, y multiplicada por la matriz que se encuentra en la cima de stack de matrices correspondiente.

En este trabajo se va partir del supuesto de que se dispone de un hardware gráfico con capacidad para realizar multiplicaciones de matrices 4x4, siendo éste un requisito exigible a cualquier hardware gráfico que quisiese dedicarse a una aplicación que incorporase múltiples actores virtuales. En el caso de que hardware gráfico empleado no dispusiese de esta capacidad, esta operación podrá ser realizada igualmente en la CPU, pero con la consecuente perdida de rendimiento.

6.2.3 Situación de los Grafos de Escena actuales respecto al procesado de las matrices de transformación.

El principal inconveniente de realizar operaciones con matrices en el hardware gráfico es que el resultado dichas operaciones reside en el propio hardware gráfico, y el acceso a esta información por parte de la CPU presenta un tiempo de respuesta muy lento[2]. En el caso de que la CPU necesite conocer el resultado de una multiplicación de matrices, resulta más rápido realizar directamente la multiplicación en la propia CPU, que enviársela al hardware gráfico para que realice la multiplicación, y luego hacer una consulta sobre su resultado. Ésta es una de las razones por las cuales los grafos actuales realizan todas las operaciones de multiplicación de matrices directamente en la CPU, desaprovechando la ventaja aportada por el hardware gráfico.

Sin embargo, como se verá, es posible definir métodos que logren que gran parte del procesado de los actores virtuales sea realizado sin necesidad de que la CPU conozca el resultado de la multiplicación de las matrices, y, por tanto, posibilitan la utilización del hardware gráfico con este fin, y evitan el cuello de botella existente a nivel de este tipo de operaciones.

Durante el procesado de un grafo de escena tradicional existen varias operaciones que necesitan conocer la posición absoluta de los nodos:

- Cálculos relacionados con *Culling*. Es necesario conocer la posición absoluta de los *bounding spheres* de los nodos respecto a la pirámide de visión.
- Cálculos relacionados con la *Gestión de Niveles de Detalle*. Es necesario conocer la posición del punto de referencia del LOD para calcular su distancia con respecto al punto de visión.
- Necesidades específicas de la aplicación, tales como realizar detección de colisiones entre objetos, cálculos de cinemática inversa, etc.

La necesidad de conocer la posición absoluta de los nodos, fuerza a que la CPU tenga que conocer los valores de las matrices de transformación, y, por tanto, a que las operaciones de multiplicación de matrices no puedan ser realizadas en el hardware gráfico. En los *grafos de escena* tradicionales (como OpenGl Performer), estas multiplicaciones de matrices son, siempre, realizadas en la CPU, de esta forma

[2] El hecho de que la pipeline gráfica este preparada para trabajar de forma direccional hace que cualquier orden de consulta sobre el estado interno del hardware gráfico -por ejemplo la rutina glGetFloatv(GL_MODELVIEW_MATRIX, mat) que retorna la matriz de modelview actual - presente un tiempo de respuesta del orden de milisegundos.

se tiene acceso a los valores de la posición absoluta de los nodos, pero esto ocurre a costa de desaprovechar la rapidez del multiplicador de matrices existente en el hardware gráfico. Esta forma de trabajo puede resultar adecuada para *grafos de escena* tradicionales centrados en la representación de escenas estáticas, pero, tal y como se ha visto en el apartado anterior, supone un cuello de botella si se aplica en una simulación que incorpore múltiples actores virtuales. En este trabajo se proporcionan métodos de gestión de LOD y culling alternativos, los cuales permiten procesar gran parte de la escena sin necesidad de que la CPU conozca las posiciones absolutas de los nodos, de tal modo que resulte posible aprovechar la rapidez del hardware de multiplicación de matrices.

Siempre existirán circunstancias en las cuales sea necesario conocer la posición absoluta de alguna geometría del actor. Estos casos serán considerados como especiales, y bastará con colocar una marca especial sobre los nodos *Skeleton* cuya posición absoluta necesita ser conocida (el campo *trackSkeleton* de la estructura de datos del nodo *Skeleton,* se emplea con esta finalidad). La activación de este flag, fuerza a que las matrices de todos los nodos *Skeleton* que hay entre dicho nodo y su nodo *Actor* correspondiente, sean calculadas en la CPU.

6.2.4 Conclusiones.

En una aplicación de simulación que contenga múltiples actores virtuales, es necesario realizar muchas multiplicaciones de matrices, lo cual afecta considerablemente a la velocidad de la simulación, y puede suponer un cuello de botella. Este cuello de botella puede ser evitado empleando la capacidad de algunos tipos de hardware gráfico para realizar este tipo de operación, y modificando el funcionamiento tradicional de los grafos de escena para que realmente puedan aprovechar esta ventaja. La situación tanto del hardware gráfico actual como de los grafos de escena no es la más adecuada: Los fabricantes de hardware gráfico actual no se preocupan de implementar multiplicadores de matrices, y aún en el caso de que esté soportada, los grafos de escena actuales no aprovechan totalmente esta ventaja característica.

En la actualidad, solamente algunas tarjetas gráficas son capaces de realizar multiplicaciones de matrices, sin embargo, es previsible que en un corto periodo de tiempo, las tarjetas de gran consumo incorporen esta capacidad, e incluso, que dada la facilidad con la dicha operación puede ser paralelizada el factor de mejora con respecto a la CPU (en el caso de una Onyx2 IR dicho factor es 12), se vea incrementado. Todos los análisis realizados en este trabajo parten del supuesto de que se está empleando hardware con esta capacidad, que sería exigible a cualquier gráfica que quisiese ser en simulaciones con múltiples actores virtuales.

Respecto a los grafos de escena, se han de proponer nuevos métodos de gestión de nivel de detalle y culling que sean capaces de actuar, sin necesidad de acceder a los resultados de las multiplicaciones de matrices. Estos métodos son introducidos en los siguientes apartados, en ellos se definen en detalle la forma en la que actúan estos métodos de culling y gestión de LOD específicos para actores propuestos por este trabajo.

6.3 Método de culling específico para Actores Virtuales.

El procesado tradicional de una escena 3D para tiempo real consiste en aplicar primero todas las transformaciones de los objetos, realizar a continuación una selección de los objetos que estén dentro de la pirámide de visión (culling), y enviar dichos objetos al hardware gráfico para su dibujado. Este esquema de trabajo, orientado a la reducción de polígonos resulta adecuado para una aplicación de simulación tradicional, pero desperdicia gran cantidad de recursos en el caso de una simulación que presente elementos con comportamientos complejos, o con muchas matrices de transformación. En el caso de una escena con actores virtuales se dan ambas circunstancias.

El procesado de un grafo de escena tradicional se divide en tres etapas que actúan de forma consecutiva. Dichas etapas, presentadas de un modo muy simplificado, son las siguientes:

1. Se ejecuta el código que aplica todas las modificaciones necesarias sobre el *grafo de escena*, típicamente estos cambios consisten en modificar los valores de los nodos *DCS*, seleccionar ramas hijas de los nodos *SWITCH,* etc.
2. Se recalcula de forma jerárquica las *bounding spheres* de los nodos de la escena que han sufrido algún tipo de modificación (en una escena tradicional la gran mayoría de los nodos permaneces estáticos ,y , por tanto, no necesitan realizar esta operación). Se seleccionan los objetos cuya *bounding sphere* está dentro de la pirámide de visión. Se seleccionan las ramas adecuadas de los nodos *LOD* en función de su distancia a la cámara, y se crea una lista con los objetos que han de ser dibujados.
3. Esta lista es enviada al hardware gráfico para su dibujado.

En este tipo de grafos de escena, existe una clara diferencia entre el proceso que gestiona el comportamiento de los objetos de la escena (comúnmente llamado APP) y el proceso que determina que polígonos han de ser enviados al hardware gráfico (comúnmente denominado CULL).

Supongamos un grafo de escena que dispone de nodos *Actor* y *Skeleton*, y que es procesado siguiendo este esquema tradicional. En estas circunstancias, la fase de dibujado de una escena con múltiples actores virtuales vendría precedida de las siguientes etapas:

Etapa 1. Ejecución del código necesario para calcular los valores de los campos de los nodos *Actor* y *Skeleton* de todos los actores existentes en la escena (gestión del comportamiento del actor).

Etapa 2. Generación de las matrices de transformación de los nodos Actor y Skeleton a partir de los valores calculados en la etapa anterior.

Etapa 3. Las matrices de transformación generadas por cada nodo *Actor* y *Skeleton* han de ser

multiplicadas de forma adecuada durante el recorrido del grafo de escena. El modo de operación del proceso de culling tradicional fuerza a que estas multiplicaciones hayan de ser realizadas en la CPU.

Etapa 4. Es necesario hacer un recálculo jerárquico de los *bounding spheres* de los nodos de la escena. La *bounding sphere* de cada nodo es calculada a partir de las *bounding spheres* de sus nodos hijos, los nodos terminales del grafo de escena calculan sus *bounding spheres* a partir de su información geométrica.

Etapa 5. Se han de realizar comprobaciones de intersección entre las distintas *bounding spheres* con la pirámide de visión. Con ello se determinan cuales son las partes de los actores que están dentro de la pirámide de visión, y se almacenan en la lista de objetos que serán enviados al hardware gráfico.

Cualquiera de estas etapas consume una importante cantidad de tiempo de CPU. Si esta forma de procesado tradicional, se aplicase sobre una escena con múltiples actores virtuales, resultaría que la gran mayoría de los costosos cálculos realizados en las cuatro primeras etapas, serían desperdiciados por la simple razón de han sido hechos sobre actores que están fuera de la pirámide de visión, y, por tanto, son eliminados en la etapa final. Resulta imprescindible que estos cálculos sean realizadas únicamente sobre aquellos actores que tienen una probabilidad alta de ser dibujados. Esto hace necesario, que el culling, y la gestión de comportamiento de los actores, hayan de ser realizados de forma conjunta, y no como dos etapas independientes (APP y CULL), tal y como hacen los *grafos de escena* tradicionales.

En un grafo de escena tradicional no se puede saber si un actor está, o no, dentro del frustum hasta que se ha calculado su comportamiento, generadas y aplicadas las matrices de los nodos y recalculadas jerárquicamente las *bspheres*. El método de culling ha de ser modificado de tal modo que se pueda determinar si un actor está fuera de la pirámide de visión sin necesidad de realizar estos costosos cálculos. La forma de conseguir esto consiste en hacer que el nodo *Actor* emplee *bounding spheres* estáticas que han sido fijadas en una fase previa a la simulación.

La utilización de estas *bounding spheres* estáticas permite que se pueda estimar si el actor está dentro o fuera del frustum sin necesidad de gestionar su comportamiento, ni tener que realizar ningún tipo de operación con sus nodos *Skeleton*. En la gestión de un actor virtual se emplearan tres tipos de *bounding spheres*, dos de ellas de tipo estático, una que engloba al actor, y otra que engloba al actor en una determinada área de movimiento, y un tercer tipo de *bounding sphere* dinámico similar a las empleadas en los grafos de escena tradicionales. Éstas últimas *bspheres* rodean a las distintas geométricas que constituyen al actor internamente, y se calculan jerárquicamente de la forma tradicional.

La utilización de este tipo de *bounding volumes* estáticas, y el hecho de que el comportamiento del actor se gestione después de haber comprobado si realmente está dentro de la pirámide de visión, hace que la organización tradicional del trabajo en forma de pipeline APP->CULL->DRAW, necesite ser modificada.

En los siguientes apartados, se detallan los distintos tipos de *bounding spheres* empleados en el nuevo tipo de culling para actores virtuales, se presentan las distintas fases en las que se divide el procesado de los actores, se muestra un ejemplo de como interactúan los distintos tipos de *bspheres* con la gestión del comportamiento y el culling en los actores virtuales, se analiza con detalle cuales son las circunstancias en las cuales resulta rentable realizar comprobaciones con las *bspheres* de los elementos que componen internamente al actor, y, por último, se describe como se realiza la integración entre el culling en fases de los actores virtuales, y el método de culling tradicional empleado para el resto de los objetos de la escena.

6.3.1 Distintos tipos de Bounding Spheres empleados el Culling de los Actores Virtuales.

Tal y como se ha introducido en el apartado anterior, en la gestión de los actores virtuales se emplean tres tipos de *bounding spheres*. Veamos ahora sus características con más detalle.

La ***Sklsroot Bsphere*** es una esfera centrada en el *Skeletons Root* del actor, con un radio suficiente como para englobar al actor virtual completo, independientemente de la postura que éste adopte. El radio de esta esfera en la actualidad es asignado de forma manual, si bien podría ser calculado mediante procedimientos automáticos.

La ***Refpoint Bsphere*** es una esfera que es capaz de englobar a un actor virtual y su movimiento. A diferencia del *Sklsroot Bsphere* que tiene un tamaño fijo para cada tipo de actor, el tamaño y posición de la *Refpoint Bsphere* depende del tipo de movimiento que esté realizando el actor. Este tipo de *bounding sphere* tiene sentido en el contexto de actores en los que el movimiento del *Skeletons Root* con respecto al *Refpoint* se desarrolla dentro de un área restringida. Éste sería el caso de un actor que esté ejecutando una determinada tabla de *keyframing*, o ejecutando algún tipo de movimiento que se desarrolle dentro de un área limitada (por ejemplo el movimiento de un niño balanceándose en un columpio).

Las ***internal Bspheres*** son el conjunto de *bspheres* que engloban las distintas geometrías que representan al actor virtual. La posición y tamaño de estas *bspheres* se modifica dinámicamente en función del cambió de los valores de los grados de libertad. Estas *bspheres* se organizan de forma jerárquica, así, por ejemplo, en el caso de un actor humano, se calcularía primero la *bsphere* correspondiente al pie, a continuación se calcularía la *bsphere* que sería la suma de la *bsphere* de la pantorrilla y la *bsphere* del pie, a continuación la suma de ésta última con la *bsphere* del muslo. Así sucesivamente, hasta llegar a tener una *bsphere* adaptada a la geometría global del actor. Esta última *bsphere* se adaptaría a la geometría global del actor de una forma similar a como lo hace la *Sklsroot Bsphere*, pero como se ha visto, para poder calcularla es

necesario que se hayan calculado los valores de los grados de libertad del actor, y, que, además, se hayan realizado distintas operaciones de transformación y composición de *bounding spheres.*

La comprobación de culling en los actores se realiza en forma de cascada: Primero se comprueba la intersección de la *Refpoint Bsphere* con la pirámide de visión, y solamente en el caso de que esté parcialmente dentro, se realiza la siguiente comprobación que consiste en calcular la posición del *Skeletons Root* y analizar la posición de la *Sklsroot Bsphere* respecto a la pirámide de visión. Solamente en el caso de que la *Sklsroot Bsphere* estuviese parcialmente dentro sería necesario entrar a realizar comprobaciones con las *Internal Bspheres*.

En la *Figura 6-1* se pueden observar los distintos tipos de *bspheres* empleados en el procesado de un actor virtual, en ella se muestra representa un actor ejecutando una pirueta a partir de los valores almacenados en una tabla de *keyframing*. La *Refpoint Bsphere* aparece representada en forma de una circunferencia en línea negra discontinua, se puede observar como engloba todas las posibles posiciones en las que puede estar el actor durante la ejecución de esa secuencia de movimiento. La *Sklsroot Bsphere* aparece representada como tres circunferencias de color negro que engloban al actor independientemente de la postura en la que se encuentre, y que tienen como centro su *Skeletons Root.* Por último, las *Internal Bspheres* aparecen representadas mediante las distintas circunferencias de color gris situadas en torno a las diferentes geometrías que componen el actor.

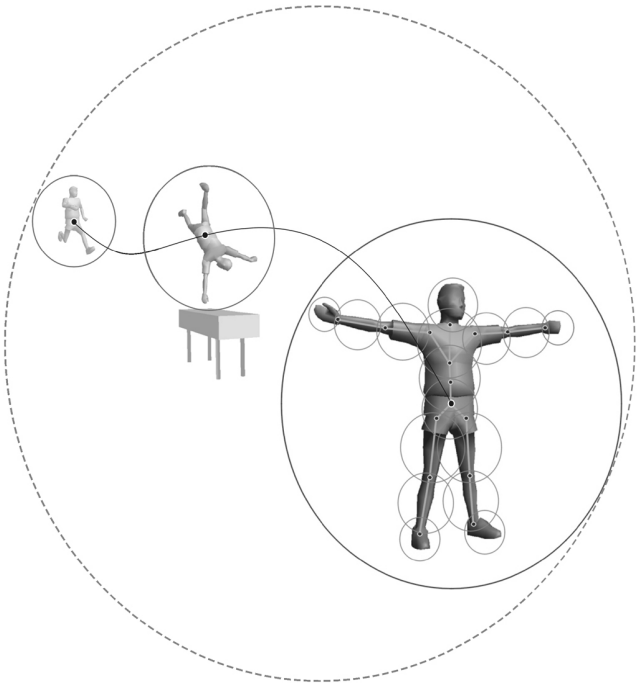


Figura 6-1. Representación de los 3 distintos tipos de *bounding spheres* empleados en la gestión de culling de un actor virtual.

6.3.2 Organización en Fases del Culling de Actores.

Los cálculos implicados en la gestión del comportamiento de un actor virtual pueden consumir una cantidad muy elevada de recursos (cálculos de dinámica y cinemática inversa, expresión facial, visión artificial...). Resulta imprescindible que las operaciones de gestión del comportamiento de los actores sean realizadas únicamente sobre aquellos que realmente vayan a ser dibujados. En este apartado se establece una división de la gestión del comportamiento en tres bloques, y también se define un método por el cual la gestión del culling de un actor virtual es realizada en distintas fases consecutivas, las cuales actúan bloqueando la gestión de comportamiento de una forma coherente con el culling.

La gestión del comportamiento de un actor virtual puede considerarse dividida en tres tipos de operaciones:

- **Cálculo de las transformaciones del *Reference Point*.** La ubicación del *Reference Point* es fijada mediante la llamada a la función *refpointBehaviourFunc* del nodo *Actor*.
- **Cálculo de las transformaciones del *Skeletons Root*.** La posición del *Skeletons Root* con respecto al *Reference Point* es fijada mediante la llamada a la rutina *sklsrootBehavioiurFunc* del nodo *Actor*.
- **Cálculo de los valores de los grados de libertad**. Son los cálculos que determinan la postura del actor. Los valores de los grados de libertad son fijados mediante la llamada a la función *dofsBehaviourFunc* del nodo *Actor*.

En el caso de emplear un grafo de escena tradicional, estos tipos de operaciones serían realizados sobre la totalidad de los actores de la escena, en una fase previa a la etapa de culling (ver *Figura 6-2*).

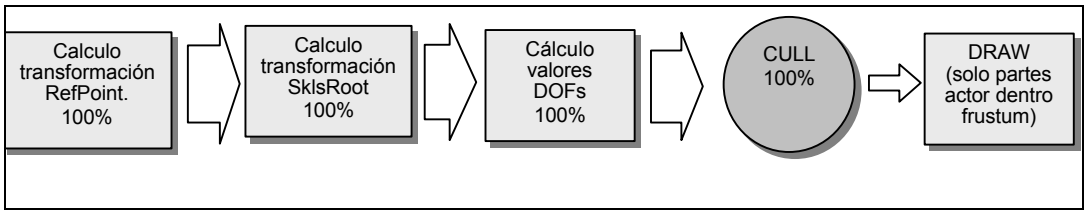


Figura 6-2. Procesado de actores mediante un grafo de escena tradicional.

En este trabajo se propone que el proceso de CULL tradicional sea modificado, de tal modo que solamente se realicen los cálculos relacionados con la gestión de comportamiento que realmente sean necesarios. Para conseguir esto, tal y como se puede observar en la *Figura 6-3*, se propone un proceso de culling dividido en tres etapas.

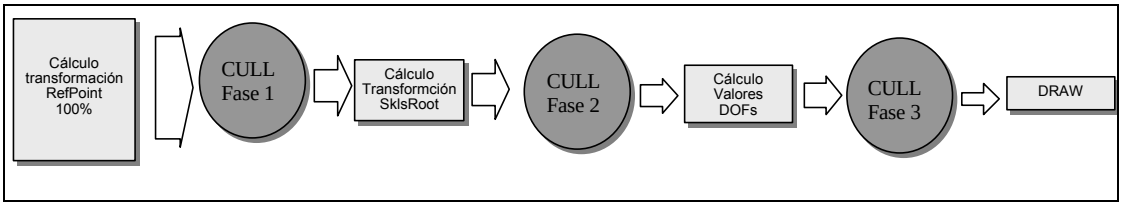


Figura 6-3. Procesado de actores empleando el método especial de culling para actores.

Según este esquema, las operaciones a realizar en cada frame sean las siguientes:

1. Se calculan los valores de las matrices *refpointMat* y *refpointAbsMat* de todos los nodos *Actor* existentes en la escena.
2. Se realiza la ***primera fase de CULL*** en la que se aplican transformaciones a las *Refpoint Bspheres,* y se comprueba la relación de dichas *bspheres* transformada con el frustum, resultando descartados todos aquellos actores cuya *Refpoint Bsphere* esté fuera de la pirámide de visión.
3. Se calculan los valores de las matrices *sklsrootMat* y *sklsrootAbsMat* de los actores que han quedado de la fase anterior.
4. Se realiza la ***segunda fase de CULL*** en la que son calculadas las posiciones absolutas de las *Sklsroot Bspheres* de los nodos *Actor*, y se comprueba la relación de dichas bspheres con el frustum, resultando descartados todos aquellos actores cuya *Sklsroot Bsphere* esté fuera de la pirámide de visión.
5. Sobre los actores que han superado la segunda fase de CULL, se realiza la gestión del comportamiento, la cual genera como resultado unos valores concretos para los grados de libertad de los nodos *Skeleton*. A partir de dichos valores se generan y aplican las correspondientes matrices de transformación.
6. Se realiza una ***tercera fase de CULL*** que determina que partes concretas del actor han de ser dibujadas, para ello es necesario realizar un recálculo jerárquico de las *Internal Bspheres* ,y las correspondientes comprobación de intersección con el frustum.
7. Se envían al hardware gráfico sólo las partes de los actores que se encuentran dentro de la pirámide de visión.

En ambos casos (*Figura 6-2* y *Figura 6-3*), el número de polígonos que llega al hardware gráfico es el mismo, sin embargo, se han reducido de forma drástica todos los cálculos de gestión de comportamiento. En una escena típica, un 70% de los actores puede ser bloqueado en la primera fase del culling de los actores, y un 10% adicional en la segunda fase, es decir, es habitual que tan sólo un 20% de los actores originales lleguen a estar en la necesidad de calcular los valores de sus grados de libertad, y tener que realizar gestiones con sus nodos *Skeleton*.

6.3.3 Ejemplo de utilización del método especial de culling para Actores Virtuales.

Para entender la forma en la que este método de culling interactúa con el procesado de los actores virtuales veamos el ejemplo mostrado en la *Figura 6-4*. En ella se muestra una escena con varios actores, cada uno de los cuales presenta una relación diferente con la pirámide de visión. En dicha figura, el frustum es representado por medio de un rectángulo, y las distintas *bounding spheres* son representadas por medio de circunferencias coloreadas en dos tonos, el tono oscuro para indicar que se comprueba su intersección con el frustum, y el tono claro para indicar que no. De un modo análogo, las partes de los actores que van a ser dibujadas son representadas en tono oscuro y las que no, en tono claro.

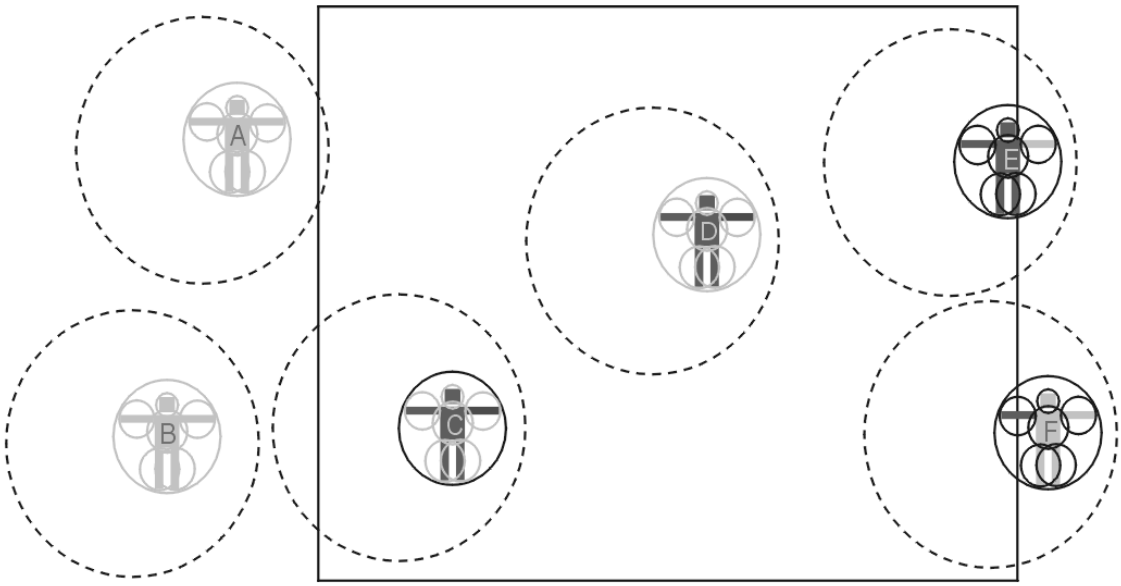


Figura 6-4. Proceso de culling sobre actores con distintas posiciones respecto al frustum.

Se puede observar como se hace una comprobación de intersección en el caso de todas las *Refpoint Bspheres*, también se puede observar como solamente se realizan comprobaciones con las *Sklsroot Bspheres* de aquellos actores cuya *Refpoint Bsphere* intersecta con alguno de los márgenes del frustum (actores A, C, E y F), mientras que no es necesario realizar ninguna comprobación con los actores cuya *Refpoint Bsphere* está totalmente dentro del frustum (actor D) o totalmente fuera (actor B). De un modo similar, se puede observar como sólo se entra a realizar comprobaciones de culling con las *Internal Bspheres* de aquellos actores cuya *Sklsroot Bsphere* intersecta con los márgenes del frustum (actores E y F), mientras que no es necesario en aquellos que tienen su *Sklsroot Bsphere* totalmente fuera del frustum (actor A) o totalmente dentro (actor C). Respecto a la gestión de comportamiento, en todos los actores se han de realizar los cálculos relativos a la posición del *Reference Point* para poder ubicar correctamente sus *Refpoint Bspheres*, los actores cuya *Refpoint Bsphere* está dentro del frustum o intersecta con sus laterales han de calcular la posición del *Skeletons Root* (actores A ,C ,D, E y F), y, de entre ellos, se ha de calcular los valores de los *DOFs* de los actores cuya *Refpoint Bsphere* esté total o parcialmente dentro del frustum (actores C ,D , E y F).

El caso de los actores E y F es especial, en ambos se realiza una gestión total de su comportamiento, y en ambos se realizan comprobaciones de CULL con los tres tipos de *bspheres*. La única diferencia reside en el hecho de que en el caso del actor F la fase 3 del culling determina que solamente es necesario dibujar un brazo, lo cual supone un ahorro considerable, mientras que en el caso del actor E es el contrario, puesto que tan sólo se ahorra el tiempo de dibujado de un brazo. En este último caso, puede ocurrir que propio proceso de gestión de su fase 3 de culling consuma más tiempo que el ahorrado por el hecho de no dibujar el brazo que está fuera del frustum, ésta es una circunstancia interesante que es analizada en detalle en el apartado que sigue.

6.3.4 Procesado de las Internal bounding spheres de los Actores Virtuales.

Como se ha descrito anteriormente, el nuevo método de CULL para actores virtuales está dividido en tres fases, en la última de las cuales se realizan comprobaciones con las *bounding spheres* de las distintas geometrías de las que está compuesto el actor. Esta tercera fase del CULL tiene un coste computacional considerablemente superior a las dos anteriores, existiendo circunstancias en las cuales su aplicación no resulta rentable.

Para determinar la rentabilidad de la tercera fase de CULL hay que considerar de una forma conjunta los costes de comprobación del culling, y los costes de dibujado. Los posibles costes asociados al procesado de cada una de las geometrías internas del actor son los siguientes:

- Se analiza su *bsphere* y está fuera: Coste comprobación de CULL.
- Se analiza su *bsphere* y está dentro: Coste comprobación de CULL + coste de dibujado.
- No se analiza su *bsphere* y se dibuja directamente: Coste de dibujado.

Como se puede observar, en el caso de que se haya comprobado el CULL y resulte que el objeto está dentro, y, por tanto, ha de ser dibujado, la gestión de CULL puede ser entendida como un coste adicional al proceso de dibujado. En el caso de que exista mucha probabilidad de que el objeto esté dentro de la pirámide de visión, puede resultar mejor dibujarlo directamente evitando el consumo de tiempo en comprobaciones de CULL. Dicho de otro modo, si la probabilidad de que el objeto a dibujar esté dentro de la pirámide de visión es de un 95%, existe un 95% de probabilidad de que el tiempo consumido realizando su comprobación de CULL haya sido un tiempo perdido.

En el coste asociado a la utilización de las *Internal Bspheres* en un actor virtual intervienen tres factores:

- El recálculo jerárquico de las *bounding spheres* a cada frame, lo cual requiere la transformación de los *bounding spheres* con las matrices de los nodos *Skeleton*, así como la combinación de distintas *bounding spheres*.
- La comprobación de intersección de las *bounding spheres* transformadas con el frustum.

- El coste de realización de las multiplicaciones de matrices en la CPU. El procesado de las *Internal Bspheres* hace necesario que la CPU conozca las matrices absolutas de los nodos, y, por tanto, fuerza a que las multiplicaciones de matrices sean realizadas en la CPU. La multiplicación de la matriz actual en la CPU ha de ser considerada como un coste añadido, puesto que esa misma operación puede ser realizada en el hardware gráfico de un modo mucho más eficiente.

La rentabilidad de la gestión de CULL de un objeto depende de su coste de dibujado, del coste de su gestión del CULL, y también de la probabilidad de que realmente vaya a ser dibujado. La comprobación de CULL de un objeto solamente resultaría rentable en el caso de que se cumpliese la siguiente expresión:

coste de dibujado > coste de CULL + probabilidad de dibujado ´coste de dibujado

o lo que es lo mismo:

coste de CULL < (1 - probabilidad de dibujado) ´ coste de dibujado

Esta expresión puede ser empleada como condición para controlar la activación/desactivación de la tercera fase de culling en un actor virtual. En el caso de los actores virtuales, el *coste de CULL* sería el coste derivado de la comprobación del culling con las *Internal Bspheres* del actor, éste es un valor fijo para cada actor, que puede ser calculado en una fase de preproceso. El *coste de dibujado* representa el coste computacional del dibujado de todas las geometrías que componen al actor, y depende, por tanto, de la complejidad geométrica del actor, y de su ocupación en pantalla (puede ser estimada a partir de la relación entre su *Sklsroot Bsphere* y el frustum). La *probabilidad de dibujado* representa la probabilidad media de que cada una de las distintas geometrías que componen el actor, vayan a ser realmente dibujadas. Es un valor que puede ser estimado a partir del cálculo de la porción de la *Sklsroot Bsphere* que está dentro de la pirámide de visión. Desde el punto de vista práctico, el actor virtual dispondrá de una función que empleará las características del frustum para poder determinar si resulta o no rentable realizar comprobaciones de culling con sus *Internal Bspheres*.

A pesar de que los cálculos relacionados con la fase 3 de culling de los actores pueden ser elevados, la realización de esta operación resulta claramente adecuada en aquellos casos en los que solamente una pequeña porción del actor está dentro de la pirámide de visión, (caso del actor F representado en la *Figura 6-4*). En estos casos, si no se aplicase la esta tercera fase de CULL, sería necesario dibujar el actor virtual entero, lo cual resultaría especialmente penoso si el actor estuviese ocupando gran parte de la imagen final, o fuese geométricamente muy complejo.

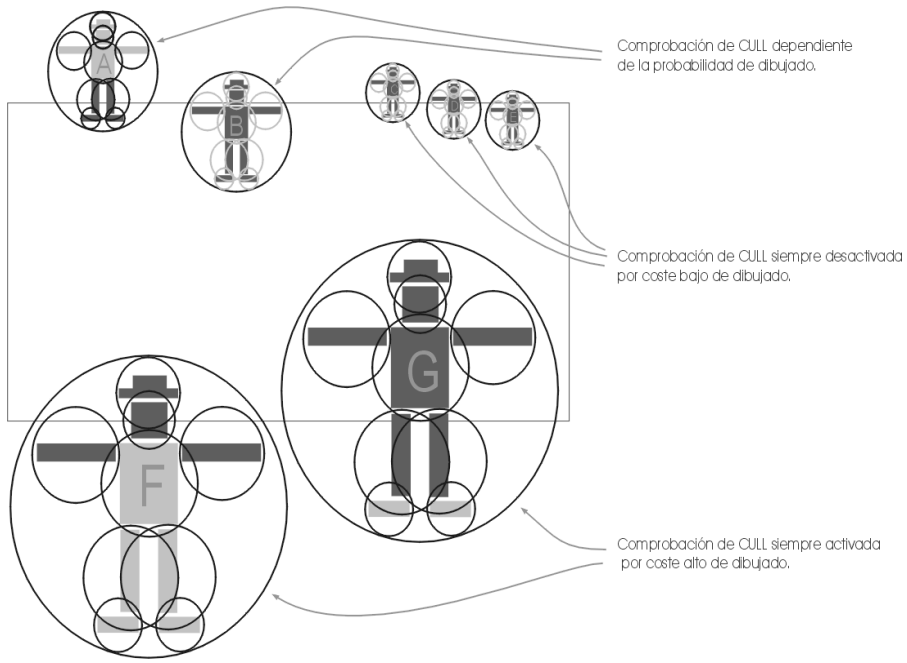


Figura 6-5. Aplicación de CULL con las *Internal Bspheres* en función de la relación existente entre la *Sklsroot Bspheres* y el frustum.

En la *Figura 6-* se muestra un ejemplo de como la relación entre la *Sklsroot Bsphere* y el frustum determina si es o no conveniente realizar comprobaciones con las *Internal Bspheres*. Los actores C, D y E presentan un coste de dibujado tan bajo que resulta más rápido dibujarlos directamente sin realzar ninguna comprobación con sus *Internal Bspheres*. En el otro extremo los actores F y G, presentan un coste de dibujado muy alto y esto hace que siempre resulte adecuado realizar la comprobación de CULL independientemente de la proporción de actor que esté dentro del frustum. El caso de los actores A y B es el más ilustrativo, ambos tienen un coste de dibujado similar, pero, el hecho de que el actor B tenga una probabilidad alta de dibujado (una gran parte de su *Sklsroot Sphere* está dentro del frustum), hace que se tome la decisión de dibujarlo entero, sin realizar ninguna comprobación del CULL con sus *Internal Bspheres*.

6.3.5 Interacción entre el método de culling especial para Actores Virtuales y el culling genérico de la Escena.

En los apartados anteriores, se analizado cual sería la forma más adecuada de llevar a cabo las operaciones relacionadas con el culling de los actores virtuales, dándose por supuesto que el resto de los elementos de la escena que no son actores virtuales son procesados mediante el método de culling tradicional. En la practica es posible que ambos métodos de culling presenten interdependencias: El culling de los actores virtuales puede afectar a elementos de la escena, y el culling de elementos de la escena pueden afectar a los actores virtuales. Esta interacción entre el culling de los actores virtuales y el tradicional puede ser de dos tipos:

- Un trozo de grafo de escena depende de un actor virtual. Éste sería el caso de un actor que decide sujetar un objeto de la escena (una herramienta por ejemplo).

- Un actor depende directamente de algún nodo genérico del grafo de escena, esto puede ocurrir en el caso de que exista alguna estructura superior que agrupe un conjunto de actores (por ejemplo una bandada de pájaros), o a un grupo de actores y un conjunto de elementos de escena tradicionales (por ejemplo un edificio con todas su estructura y muebles y con varias personas en su interior, o un vehículo con varios ocupantes en su interior).

En el primer caso, cuando se haga una comprobación de culling con la *Refpoint Bsphere* o la *Sklsroot Bsphere* del actor, ha de tenerse en cuenta que esa operación ha de afectar también a la herramienta que sostiene el actor. Esto se consigue incrementando el radio de la *Sklsroot Bspheres* de forma que incluya al actor y el objeto que depende de él, y aumentando también, de forma coherente, el radio de las *Refpoint Bspheres*.

En el segundo caso, el culling tradicional afecta al procesado de los actores, así, en el ejemplo del vehículo, el culling del habitáculo determina si los actores que hay en su interior han de ser procesados o no.

6.3.6 Análisis del coste relacionado con la gestión del Culling.

En este apartado se va realizar un análisis sobre el coste asociado a la gestión de culling de los actores virtuales, y se va a establecer una comparación con respecto a la forma de operar del culling tradicional. El objetivo es mostrar de una forma cuantitativa la ventaja proporcionada por el método específico propuesto en este trabajo.

Las operaciones relacionadas con el CULL se realizan a tres niveles, a nivel de las *Refpoint Bspheres,* a nivel de las *Sklsroot Bspheres*, y a nivel de las *Internal Bspheres*. Supongamos una escena formada por varios actores virtuales del mismo tipo, y cuyas *bspheres* estáticas tienen un tamaño y distribución tales que la *Refpoint Bsphere* de un porcentaje de ellos intersecta con los laterales de frustum (*fRefpointBsphIsect*), y ocurre lo mismo con la *Sklsroot Bsphere* de otro porcentaje distinto de ellos (*fSklsrootBsphIsect*). Supongamos que siguiendo los criterios del apartado 6.3.4, solamente un porcentaje (*fCheckIntBsphs*) del total de actores tienen una relación con el frustum, que hace adecuado realizar comprobaciones de CULL con sus *Internal Bspheres*.

Para poder realizar el análisis de una forma más sencilla vamos a suponer una escena formada por 100 actores (nActors = 100), en la que existen 20 actores cuya *Refpoint Bsphere* intersecta con el frustum (fRefpointBsphIsect = 0.2), 8 actores cuya *Sklsroot Bsphere* intersecta con el frustum (fSklsrootBsphIsect = 0.08) y 2 actores que tienen una relación con el frustum tal que es adecuado realizar comprobaciones de CULL con sus *Internal Bspheres* (fCkeckIntBsphs = 0.02).

En dicha escena es necesario realizar la comprobación de intersección de las *Refpoint Bspheres* de los 100 actores con el frustum, obteniéndose como resultado que 70 están totalmente fuera, 10 totalmente dentro y

20 intersectan con los laterales del frustum. Se chequea la relación de la *Sklsroot Bsphere* de estos últimos 20 actores con el frustum, dando como resultado que 2 están totalmente fuera, 10 totalmente dentro, y 8 intersectan con el frustum. En el caso de emplear el culling tradicional habría que chequear las *Internal Bspheres* de estos 8 actores. En el caso de emplear el método de culling específico de actores, y suponiendo un factor *fCheckIntBsphs* = 0.02, bastaría hacerlo con 2 de ellos. Para estimar el coste de las comprobaciones con las *Internal Bspheres* supongamos que cada actor virtual tiene el doble de nodos que de puntos de articulación, así, en el caso de un actor con 40 puntos de articulación (*nSkelsPerAct*), tendríamos que comprobar intersecciones con 80 *bspheres* (*nBsphsPerAct*). La comprobación jerárquica de culling hace que en realidad no sea necesario llegar a chequear todas las ramas del subgrafo que constituye al actor virtual, pudiéndose suponer que de media, es necesario procesar tantas *Internal Bspheres* como puntos de articulación tenga el actor.

El número de operaciones de recálculo de *bspheres* puede ser expresado del siguiente modo:

	culling tradicional	culling actores
Refpoint Bspheres	nActors	0 (ha sido precalculada)
Sklsroot Bspheres	nActors	0 (ha sido precalculada)
Internal Bspheres	nActors* nBsphsPerAct	nActors* fChekIntBSphs*nBsphsPerAct

Es decir:

- culling tradicional -> nActors*(1+1+nBsphsPerAct)
- culling de actores -> nActors *fCheckIntBsphs* nBsphsPerAct

De un modo similar, el número de operaciones de intersección entre *bspheres* y frustum sería:

	culling tradicional	culling actores
Refpoint Bspheres	nActors	nActors
Sklsroot Bspheres	nActors*fRefpointBsphIsect	nActors*fRefpointBsphIsect
Internal Bspheres	nActors*fSklsrootBsphIsect*nSkelPerAct	nActors*fCheckInBsphs* nSkelsPerAct

Es decir:

- culling tradicional -> nActors * (1 + fRefpointBsphIsect +sklsRootBsphIsect*nSkelsPerAct)
- culling de actores -> nActors * (1 + fRefpointBsphIsect + fCheckIntBsphs* nSkelsPerAct)

Si en estas expresiones empleamos los valores de la escena tomada como ejemplo, obtenemos:

- coste culling tradicional = 8200 recálculos de bsphere+520 comprobaciones de intersección.
- coste culling actores = 160 recálculos de bsphere +200 comprobaciones de intersección.

En función de estos resultados, se puede observar como los cálculos relacionados con el recálculo de *bspheres* son unas 50 veces más rápido, y el coste de comprobación de intersecciones con el frustum unas 2.5 veces más rápido.

6.4 Métodos de gestión de LOD específicos para actores.

El nivel de detalle dentro de un *grafo de escena* es usualmente gestionado por unos nodos especiales denominados *LOD* (Level Of Detail). Un *LOD* es un nodo que presenta varias ramas hijas, cada una de las cuales representa al mismo objeto con un nivel de complejidad diferente. Cuando durante el recorrido del grafo de escena, se llega a uno de estos nodos, se calcula la distancia existente entre el punto de vista y un punto de referencia que presenta el *LOD* (punto central del *LOD*), y, en función de esa distancia (y algún otro criterio), se selecciona cual es la rama hija que ha de ser dibujada. En un grafo de escena tradicional toda la gestión de nivel de detalle es realizada mediante este tipo de nodos.

Las bases de datos empleadas en una simulación tradicional suelen tener pocas transformaciones, y una gestión del comportamiento muy simple, esto hace que su gestión de nivel de detalle se reduzca a realizar un control del número de polígonos que son enviados al hardware gráfico. En el caso de los actores virtuales, es necesario realizar este tipo de control, al que denominaremos ***nivel de detalle geométrico,*** pero también es necesario realizar una gestión del ***nivel de detalle topológico*** que actúe sobre el número de transformaciones que aplica un actor, y una gestión de ***nivel de detalle comportamental*** que permita hacer que el coste computacional de la gestión del comportamiento se reduzca con la distancia. En los siguientes apartados se detalla la forma en la que estos tres tipos de gestión de nivel de detalle actúan sobre los actores virtuales.

La selección del nivel de detalle adecuado en cada momento es determinada función de la distancia a la que el actor se encuentre de la cámara, pero, además, cada actor virtual tendrá un campo adicional, que actuará sobre el criterio de selección de nivel de detalle, permitiendo aumentar la calidad con la que se representan aquellos actores que están desempeñando algún tipo de actividad de especial importancia para el usuario.

6.4.1 Gestión de Nivel de Detalle geométrico.

La gestión del nivel de detalle geométrico de un actor virtual podría ser realizada de varios modos. En una primera aproximación, se podría pensar que bastaría con disponer de 3 o 4 modelos diferentes del mismo actor (cada una de ellas correspondiente a un diferente nivel de detalle), produciéndose el cambio de un nivel de detalle al siguiente de forma simultanea en todas las partes del cuerpo del actor. Sin embargo, esta aproximación inicial, no tiene en cuenta que la representación poligonal de un actor virtual está formada por distintas geometrías, y no todas ellas tienen la misma importancia visual, así, para la representación de cabeza de un actor humano puede ser adecuado emplear cinco niveles de detalle diferentes, mientras que su antebrazo puede ser correctamente representado mediante sólo tres. De un modo coherente, también las distancias a las que se realiza la transición entre niveles pueden ser distintas. Esto hace necesario que el nivel de detalle con el que se representa cada parte de un actor virtual pueda

ser controlado por separado, y, por tanto, que las distintas partes del actor puedan ser gestionadas por nodos LOD independientes, que puedan tener diferente número de hijos, y distancias de transición.

La utilización de nodos LOD tradicionales para gestión del nivel de detalle de las geometrías que definen un actor virtual, puede suponer un excesivo coste computacional. Esto es debido a que cada nodo LOD necesita calcular en cada frame la distancia existente entre su punto central y el punto de vista. Este hecho presenta dos inconvenientes: por un lado hace necesario conocer la posición absoluta del punto central del LOD, lo cual, como ya se ha visto, fuerza a que las operaciones con matrices de transformación hayan de ser realizadas en la CPU, y por otro la propia operación de cálculo de distancia tiene un coste computacional que ha de ser tenido en cuenta.

Ambos inconvenientes pueden ser evitados observando que los puntos centrales de todos los nodos LOD que dependen de un mismo actor virtual, siempre permanecen próximos. Así, si la distancia entre el punto de vista, y el punto central del nodo LOD que se corresponde con la cabeza de un actor humano es de 50 metros, la distancia entre el punto central del nodo LOD correspondiente a su antebrazo estaría en un margen de 50 ±1 metro. Todos los nodos LOD de un actor virtual podrían emplear la misma distancia, cometiendo con ello un error muy pequeño. La organización del actor virtual, indica que esta distancia única ha de ser tomada con respecto al *Skeletons Root* del actor. Esta distancia va a ser empleada por todos los nodos LOD que dependen jerárquicamente de ese actor virtual, evitando con ello que sea necesario realizar múltiples cálculos de distancia por cada actor, y evitando sobre todo, que las operaciones con matrices hayan de ser realizadas en la CPU.

La distancia existente entre el *Skeletons Root* y el punto de vista es almacenada en un campo del nodo *Actor*, y es empleada por todos los nodos LOD que dependen del actor virtual, y también en la gestión del nivel de detalle topológico y comportamental.

6.4.1.1 Ampliación del funcionamiento de los nodos LOD tradicionales.

Cuando durante el proceso de recorrido de un grafo de escena tradicional se llega a un nodo LOD, éste calcula internamente su distancia a la cámara, y a partir de ella decide cual es la rama hija a dibujar. Esta forma de trabajo tiene los inconvenientes indicados en el apartado anterior, y cuya solución consiste en realizar un único cálculo de distancia por cada actor virtual y hacer que sea empleada por todos los nodos LOD que dependen jerárquicamente de él. Para que este mecanismo pueda ser aplicado es necesario que la forma de funcionamiento de los nodos LOD sea ligeramente modificada, para ello se ha definido una estructura de datos auxiliar, de nombre ***vaLod*** que sustituye a los nodos LOD tradicionales. La estructura *vaLod* tiene un puntero al nodo actor del que depende, de este modo, emplea el valor de distancia a la cámara almacenado en el campo *eyeDistance* del nodo *vaActor*.

6.4.2 Gestión de Nivel de Detalle topológico.

Un actor virtual suele estar formado por un número considerable de nodos *Skeleton*, sin embargo, en actores que estén muy lejos de la cámara, el efecto de alguno de estos nodos *Skeleton* resulta inapreciable, y, en consecuencia, su gestión supone un desperdicio de recursos de cálculo. Es necesario disponer de un método mediante el cual la complejidad topológica de un actor pueda ser reducida a medida que su distancia a la cámara aumenta. La solución propuesta en este trabajo consiste en hacer que cada nodo *Skeleton* tenga una distancia a partir de la cual deje de ser procesado, con lo cual ni se evalúan sus grados de libertad, ni se genera su matriz, ni se recorre el subgrafo que depende de él.

La necesidad de realizar este tipo de gestión resulta evidente si tomamos como ejemplo el caso de las manos de un actor humanoide: En la definición de la estructura articulada de una mano es necesario emplear 16 nodos *Skeleton* (3 para cada uno de los dedos y uno adicional para la articulación de la muñeca), sin embargo, no tiene sentido realizar ningún tipo de gestión con los dedos de un actor que se encuentra a 500 metros de distancia con respecto a la cámara. A partir de cierta distancia, las distintas geometrías que definen la mano y los dedos han de ser automáticamente sustituidos por una única geometría que representa la mano completa, y los distintos nodos *Skeleton* que representan los puntos de articulación de las falanges de los dedos han de ser desactivados.

Es necesario que exista una coordinación entre las distancias de desactivación de los nodos *Skeleton* de un actor, y las distancias de transición de los nodos *vaLod* que gestionan los niveles de detalle de su geometría. En la *Figura 6-6*, se representa el grafo de escena correspondiente a un actor humanoide muy sencillo, en el cual los nodos *Skeleton* muestran sus diferentes distancias de desactivación. Este grafo de escena permite que el actor tenga 3 niveles de detalle topológicos, el primero de ellos formado por 9 nodos *Skeleton*, el segundo formado por 4 nodos *Skeleton,* y tercero que no tiene ninguno. Los nodos con información geométrica son representados por medio de figuras humanas en las que la porción de geometría contenida en el nodo está coloreada de negro. Mediante los distintos tipos de línea, se puede observar que partes del grafo de escena son procesadas en cada uno de los niveles de detalle. Se puede suponer que el nivel de detalle 1 se da entre 0 y 10 metros, nivel de detalle 2 entre 10 y 100 metros, y el nivel de detalle 3 a partir de 100 metros. Tal y como se puede observar, el nivel de detalle 1 tiene una representación separada de la cabeza y el cuerpo, mientras que en el nivel de detalle 2, ambos elementos aparecen en la misma geometría. Para que la gestión de nivel de detalle funcione adecuadamente, es necesario que la distancia de desactivación del nodo *Skeleton* de la cabeza, coincida con la empleada para cambiar del nivel de detalle 1 al 2 en el nodo *LOD* que gestiona la representación del cuerpo

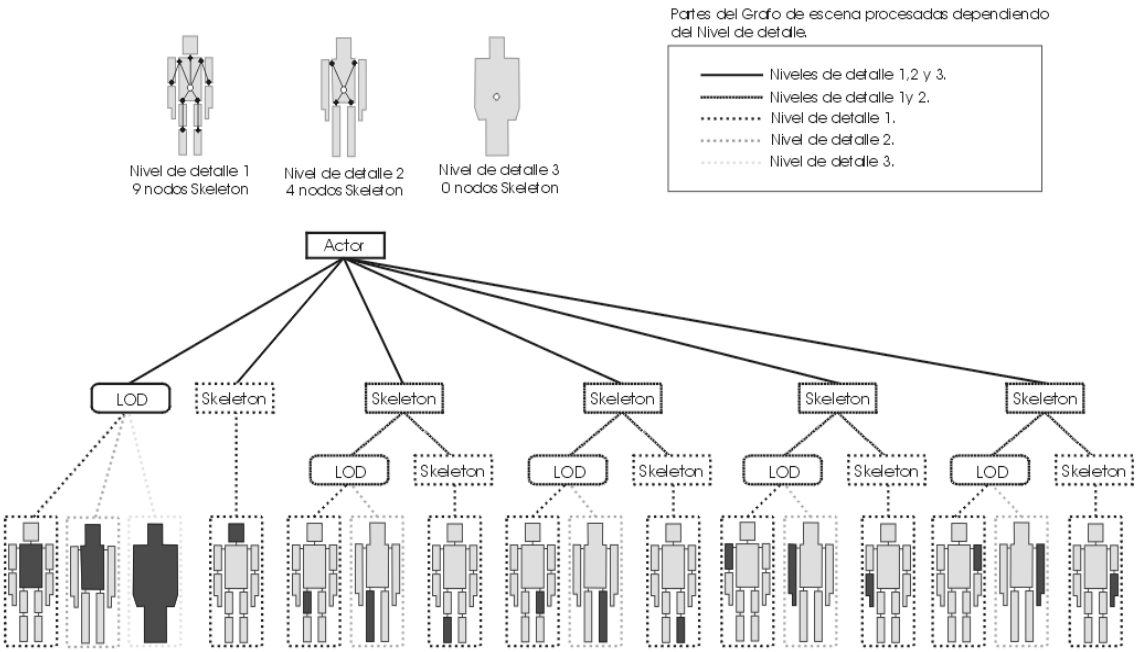


Figura 6-6. Relación entre la gestión de nivel de detalle de la geometría y de niveles de detalles topológicos de nodos *Skeleton*.

El hecho de que exista una distancia a partir de la cual un nodo *Skeleton* deje de ser procesado tiene varias implicaciones: Por un lado reduce el número de nodos *Skeleton* a ser procesados, pero, de forma paralela, afecta al número de polígonos a ser dibujados, y también a la gestión de comportamiento, puesto que no tiene sentido estar empleando un modelo de gestión de comportamiento, que realice cálculos relativos a puntos de articulación que han sido desactivados.

6.4.3 Gestión de Nivel de Detalle comportamental.

En una simulación tradicional, la gran mayoría de los recursos se emplean en el dibujado de polígonos, existiendo pocos objetos que presenten un comportamiento dinámico. En el caso de realizar simulaciones con un número elevado de elementos dinámicos (tal como puede ser el caso de un simulador de tráfico **[FER98]**), es necesario realizar una gestión de nivel de detalle que actúe sobre los costes de gestión de la dinámica de dichos elementos. En el caso de una aplicación de simulación que incorpore actores virtuales, puede ser común que más de la mitad de los recursos de CPU estén dedicados a la gestión del comportamiento de los actores (piénsese que se pueden estar gestionando comportamientos complejos que incluyan cálculos de cinemática inversa, mecanismos de visión artificial, síntesis de voz, etc.). Cuando un actor virtual está lejos, se pierde la capacidad para apreciar los detalles de su geometría, pero, de igual modo, también se pierde la capacidad para apreciar los detalles del su comportamiento. El *nivel de detalle comportamental* actúa sobre la precisión de los mecanismos que gestionan el comportamiento de un actor, haciendo que su coste computacional disminuya, a medida que su distancia al punto de vista aumenta.

Un actor virtual dispone de diferentes mecanismo que controlan distintos aspectos de su comportamiento. En el caso de un actor humano, podría existir un mecanismo responsable de la gestión de la locomoción, otro dedicado al control de actividades manuales, otro de la expresión facial, otro encargado del control de la mirada, etc. Cada uno de ellos ha de ser implementado de tal forma que sea posible tener control sobre su coste computacional. Vamos a profundizar en este tipo de gestión de nivel de detalle comportamental tomando como ejemplo la implementación de un mecanismo de control de la mirada de un actor humano.

El mecanismo encargado de la mirada de un actor no sólo ha de controlar el movimiento de los ojos, sino también ha de actuar sobre el cuello, e incluso sobre la columna vertebral. Supongamos un mecanismo de gestión de la mirada que actúa solamente sobre 3 nodos *Skeleton*: los correspondientes a cada uno de los ojos, y un tercero asociado con el punto de articulación en el cuello. Supongamos que la distancia de desactivación de nodos *Skeleton* los ojos es de 50 metros, y la distancia de desactivación del cuello es de 500 metros. Como se ha visto en el apartado anterior, esto hace necesario que el nodo *vaLod* que controla la geometría de la cabeza tuviese un modelo que se activase a los 50 metros, y en el que los ojos estén incorporados en la propia geometría de la cabeza. De igual modo, el LOD que controla la geometría del tórax, tendría un modelo que se activaría a los 500 metros, en el cual la geometría del tórax tendría incorporada una representación de la cabeza. Estos cambios topológicos producidos en un actor virtual, a causa de las distancias de desactivación de los nodos *Skeleton,* actúan directamente sobre los mecanismos de gestión de comportamiento. Así, el modelo de gestión de la mirada, puede ser desactivado para distancias superiores a 500 metros (distancia de desactivación del nodo *Skeleton* del cuello), y carece de sentido que realice ningún tipo de cálculo referente a la orientación de los ojos, para distancias superiores a 50 metros. Los cambios en la topología del actor, definen el número mínimo de niveles de detalle de comportamiento, sin embargo, el número de estos puede ser superior a los forzados por las distancias de desaparición de los nodos *Skeleton*. En el caso del mecanismo de mirada, las distancias de desactivación del cuello y los ojos fuerzan a que haya dos puntos de cambio de LOD a las distancias de 50 y 500 metros, pero pueden existir otros adicionales cuyo objetivo es tener un control más fino sobre la precisión y rapidez de los cálculos. Así, el mecanismo de gestión de mirada que estamos tratando, podría estar dividido en 5 niveles distintos:

- Mecanismo de mirada gestionado mediante cinemática inversa tradicional. Se define una cadena cinemática para cada uno ojos en la que se ven implicadas puntos de articulación de cada ojo, y, además, el cuello. El hecho de que cada ojo tenga un control independiente, permite que los ojos converjan, en el caso de estar mirando objetos muy próximos. Este nivel de detalle estaría activo entre 0 y 5 metros de distancia.
- Mecanismo con cinemática inversa similar al anterior, pero con número reducido de iteraciones, lo cual hace que sea más rápido a costa de una pérdida de precisión. Entre 5 y 25 metros de distancia.

- Mecanismo aproximado, que calcula la orientación del cuello y la cabeza sin recurrir a cinemática inversa, calculando las orientaciones para el cuello y un sólo ojo, el otro ojo emplea los mismos valores. Entre 25 y 50 metros.
- Mecanismo en el que los ojos permanecen estáticos, y sólo se modifica la orientación de la cabeza. Coincide con la distancia de desactivación de los nodos *Skeleton* de los ojos. Entre 50 y 500 metros.
- Desactivación total del mecanismo de mirada. Coincide con la distancia de desactivación del nodo *Skeleton* del cuello. A partir de 500 metros.

Como ya se ha citado anteriormente la distancia empleada para seleccionar el nivel de detalle adecuado está almacenada en el nodo *Actor*. La gestión de niveles de detalle comportamental resulta imprescindible si pretende desarrollar una aplicación en la que existan varios actores virtuales que presente un comportamiento de una complejidad media.

6.5 Multiproceso en Actores virtuales.

Cualquier aplicación de simulación, y en especial una que contenga múltiples actores virtuales consume gran cantidad de recursos, por ello, puede resultar adecuado recurrir al empleo de sistemas con varios procesadores. De un modo coherente, es necesario analizar, cual sería la forma adecuada de integrar las estructuras y métodos propuestos en este trabajo, dentro de un sistema multiprocesador. No se pretende profundizar en la forma de aplicación de estas técnicas, pero sí mostrar cuales son los aspectos de los *grafos de escena* tradicionales que resultan inadecuados, y proponer algunas posibles soluciones.

En el apartado en cual se hablaba del estado del arte del multiproceso aplicado a la informática gráfica en tiempo real, se había introducido que la formula más empleada, consiste en la división del trabajo en tres etapas organizadas en forma de pipeline. La implementación clásica de esta pipeline divide el procesado en tres etapas denominadas APP, CULL y DRAW, haciendo que cada una de ellas se procese en su propia CPU. En la etapa de APP se leen los periféricos de entrada, se calculan las posiciones de los objetos móviles, se aplican todas las modificaciones al grafo de escena (se definen las matrices de transformación de los nodos DCS, se hacen selecciones adecuadas en los nodos SWITCH, se modifican atributos relativos al color o transparencia de los objetos etc.), y se interacciona con otras posibles simulaciones conectadas a través de red. A continuación, sobre este grafo de escena, se realiza la fase de CULL, la cual engloba la selección de las porciones de escena que son potencialmente visibles, la selección de las ramas adecuadas de los nodos de nivel de detalle, y la ordenación de la geometría según diversos criterios. Por último, el proceso de DRAW, se encarga básicamente de alimentar al hardware gráfico con los objetos que han quedado de la fase anterior.

Esta forma de trabajo, es correcta para una simulación en la que existan pocas transformaciones, sin embargo, resulta totalmente ineficiente, en el caso de una simulación que incorpore múltiples actores virtuales. Si esta forma de trabajo fuese empleada para los actores virtuales, la gestión de comportamiento tendría que ser realizado en la etapa del proceso de APP, de tal modo que se estarían realizando cálculos complejos sobre actores que serían eliminados posteriormente en la fase de CULL.

Para ayudar a estructurar los procesos de una forma distinta, es necesario en primer lugar, dividir en dos bloques el trabajo realizado por el actual proceso de CULL. El proceso de CULL tradicional, también se encarga de realizar tareas de ordenación de la geometría, las cuales vamos a separar en un bloque aparte. Así tendremos un proceso de CULL encargado únicamente de realizar el culling jerárquico y la selección de niveles de detalle, y un nuevo proceso, al que se denominará **SORT,** y que estará encargado de la ordenación de la geometría.

También se va a definir una etapa nueva a la que se denominara **ACT,** que será la encargada de la gestión del comportamiento de los actores virtuales, así como del procesado de sus nodos *Actor* y *Skeleton*.

Con estas modificaciones el pipeline final pasa a estar formado por cinco etapas:

- **APP:** Realiza las modificaciones tradicionales de la escena, y, además, calcula la posición de los *Reference Point* de los Actores. No realiza ningún tipo de operación adicional sobre los actores.
- **CULL:** Genera una lista con los objetos de la escena que han de ser dibujados, y hace la *primera comprobación de CULL* con los actores virtuales, eliminando aquellos cuyas *Refpoint Bspheres* estén fuera de la pirámide de visión.
- **ACT:** Se calculan las posiciones de los *Skeleton Root* de los actores que han superado la fase anterior, se hace la *segunda comprobación de CULL,* eliminando aquellos actores cuyas *Sklsroot Bspheres* estén fuera de la pirámide de visión. Se realiza la gestión de comportamiento de los actores que han quedado, y, se calculan y aplican todas las transformaciones que dependen de los nodos *Skeleton*. Cada uno de los actores que han llegado hasta esta fase, determina si es o no necesario aplicar la *tercera fase de CULL.* El resultado final es una lista con todas las geometrías que dependen del actor, transformadas y listas para dibujar.
- **SORT:** Ordena de distintos modos las listas de geometrías generadas en las dos etapas anteriores.
- **DRAW:** Se suministra al hardware gráfico la lista de geometrías proveniente de la etapa anterior.

El hecho de que existan dos etapas más, no indica que sea necesario emplear más CPUs, simplemente es una forma diferene de estructurar el trabajo que está adaptada a las necesidades de una aplicación de simulación con actores virtuales. Al igual que ocurría en el caso en una pipeline tradicional, no se puede establecer una distribución ideal del trabajo entre las CPUs, puesto que depende del tipo de aplicación, y también del número de CPUs disponibles. En el caso de disponer de un ordenador con 2 CPUs, una configuración adecuada podría ser del tipo APPCULLACT_SORTDRAW (es decir, las etapas de APP, CULL y ACT en la primera CPU y las etapas de SORT y DRAW en la segunda), o una distribución APPCULL_ACT_SORTDRAW en el caso de disponer de 3 CPUs, o APP_CULL_ACT_SORTDRAW en el caso de 4 CPUs.

Respecto a los sistemas de procesamiento paralelo, y a arquitecturas mixtas en las cuales organización en pipeline se mezcla con CPUs que están trabajando en paralelo, es importante indicar que el procesado de los actores virtuales es fácilmente paralelizable. Este hecho es de especial importancia si se tiene en cuenta la elevada complejidad que puede llegar alcanzar la gestión del comportamiento de un actor virtual. Así, en el caso de disponer de un ordenador con 8 CPUs, sería muy adecuado emplear una estructura de tipo mixto APP_CULL_ACT(4)_SORT_DRAW, en la que hay 4 CPUs que se dedican en paralelo a gestionar, cada una de ellas, un pequeño grupo de actores. Resulta relativamente sencillo tener una estimación del coste de procesado de cada actor, lo cual facilita el paralelismo, permitiendo emplear tanto distribuciones estáticas como dinámicas, en este último caso, cada CPU iría solicitando actores a procesar por orden de complejidad descendiente.

6.6 Conclusiones.

En este capítulo, se han tratado varios aspectos del procesado de un grafo de escena, que necesitaban ser adaptados adecuadamente a las características de una simulación con múltiples actores virtuales. Estos aspectos han sido: la forma en la que se procesan las matrices de transformación, la forma en la que se realiza el culling, el modo de gestión de los niveles de detalle, y integración en un sistema multiprocesador.

En relación con el procesamiento de las matrices de transformación, se ha presentado la facilidad con la que actúan como cuello de botella, se ha mostrado como este problema puede ser evitado empleando la capacidad de algunas tarjetas gráficas para realizar multiplicaciones de matrices, y se han descrito las modificaciones necesarias a realizar sobre los grafos de escena tradicionales, para que realmente puedan aprovechar esta capacidad.

En relación con la forma de realización del culling, se ha mostrado como la gestión de culling tradicional resulta muy poco eficiente en el caso de ser empleada con objetos, que, presenten comportamientos complejos, y necesiten aplicar matrices de transformación. Para solucionar este problema, se ha propuesto un método de culling específico para actores virtuales, que es compatible con el tradicional, minimiza las operaciones de recálculo de bounding spheres, permite la utilización del hardware de multiplicación de matrices, y actúa sobre los costes asociados a la gestión del comportamiento. Este nuevo tipo de culling está caracterizado por su división en tres fases, que aparecen intercaladas con diferentes bloques de gestión de comportamiento, y por la utilización de *bounding spheres* estáticas (*Refpoint Bsphere* y *Sklsroot Bsphere)* que permiten realizar comprobaciones de *culling* sin necesidad tener que actualizar los valores de los grados de libertad del actor, ni recalcular sus *bounding spheres* de la forma jerárquica tradicional. En algunas actores cuyas *bspheres* estáticas intersectan con los laterales del frustum, puede resultar adecuado realizar comprobaciones de culling con las *bspheres* de las geometrias que lo componen internamente (*Internal bspheres*). Las especiales características de los actores virtuales, permiten emplear una fórmula que controla la activación/desactivación de esta tercera fase de culling, en función de sus costes de dibujado, gestión de culling, y ocupación en pantalla. Se he hecho una estimación teórica de la mejora introducida en relación con el método tradicional, y se ha mostrado la forma en la que se realiza la integración con el culling genérico, empleado para el resto de los elementos que no son actores.

En relación con la gestión de nivel de detalle, se ha descrito como la gestión de tipo *Geométrico,* centrada en la reducción del número de polígonos que son enviados al hardware gráfico, ha de ser ampliada con una gestión de nivel de detalle de tipo *Topológico*, que consigue que el número de articulaciones que definen un actor se reduzca don la distancia, actuando de coherentemente sobre el número de matrices de transformación procesadas por un actor, y una gestión de nivel de detalle de tipo *Comportamental*, que permita hacer que el coste de los métodos de gestión de comportamiento empleados se reduzca con la

distancia. Para la gestión del nivel de detalle geométrico se ha definido un nuevo tipo de nodo de nombre *vaLod* que es controlado desde el nodo *vaActor,* y que proporciona un control centralizado que reduce el número de operaciones de cálculo de distancia. Los tres tipos de gestión de nivel de detalle actúan de forma combinada y presentan interdependencias.

Por último, en relación con la integración en un sistema multiprocesador, se ha mostrado como la división clásica del trabajo en tres etapas APP, CULL, DRAW que se ejecutan en forma de pipeline no resulta adecuada para una simulación con actores virtuales. Como alternativa, se ha propuesto una organización basada en la utilización de una pipeline de cinco etapas de nombres APP, CULL, ACT, SORT y DRAW, que aprovecha las ventajas de las estructuras y métodos propuestos en este trabajo, y que presenta claras ventajas respecto a la aproximación clásica.

Capítulo 7. Estimación de la mejora computacional respecto a un grafo de escena tradicional.

7.1 Índice.

7.2 Introducción.

Las principales aportaciones de este trabajo mostradas en los capítulos anteriores se pueden resumir en los siguientes aspectos:

- Estandarización. Se ha analizado de cuales serían las estructuras de datos necesarias para integrar de una forma robusta, y flexible, cualquier tipo de actor virtual en una aplicación de simulación.
- Organizativas. Las estructuras y métodos proporcionados por este trabajo, actúan como capa de bajo nivel sobre la que se pueden realizar desarrollos más complejos. Permitiendo la definición de aplicaciones muy modulares, independientes de plataforma, y que pueden funcionar sobre cualquier grafo de escena que presente unos requisitos mínimos
- Computacionales. Se ha analizado cuales son los principales cuellos de botella existentes en el procesado actual de actores virtuales, y se han propuesto distintos tipos de estrategias que dan como resultado una importante mejora computacional.

Todos estos aspectos, son vitales a la hora de definir una aplicación de simulación que incorpore actores virtuales, pero, el único de ellos que puede ser estimado de una forma objetiva, es el computacional. En este capítulo, se mostrará la mejora computacional introducida por las estructuras y métodos propuestos. Para ello, se realizará un análisis exhaustivo sobre cuales son los distintos tipos de operaciones elementales implicados en el procesado de este tipo de escenas, y se obtendrán dos estimaciones sobre su coste total: La primera de ellas es una estimación cualitativa que concluye con una expresión analítica que muestra el coste del procesado en función del número de operaciones elementales realizadas. La segunda es una estimación cuantitativa en la que se emplea una escena patrón formada por un número concreto de actores con unas determinadas características. En ambos casos, se mostrarán tablas y gráficas que permitirán analizar la importancia relativa de cada operación elemental, y también realizar comparaciones entre el coste del procesado empleando las estructuras y métodos propuestos en este trabajo, y su coste si hubiese sido empleado un grafo de escena tradicional.

7.3 Diferenciación de las operaciones elementales.

Para analizar la mejora en el coste computacional, va ha ser necesario aislar los distintos tipos de operaciones, que intervienen en el procesado de los actores virtuales de la escena de simulación. A cada una de estas operaciones se les asignará una abreviatura. Estas abreviaturas serán empleadas en los cálculos y expresiones que serán realizados a lo largo del capítulo:

gms :	Generación de la matriz de un punto de articulación, puede ser considerada como la suma de dos operaciones parciales: **gms1**: cálculo de los valores de los grados de libertad. Es una operación con coste variable, que depende del tipo de gestión de comportamiento que se esté empleando (puede ir desde una simple interpolación de valores obtenidos de una tabla de *keyframes*, hasta un complejo proceso en que se requieran cálculos de dinámica, o cinemática inversa). **gms2**: generación de la matriz de transformación, a partir de los valores de los grados de libertad.
b :	Obtención de las matrices *refpointMat* y *refpointAbsMat* de un nodo Actor. También está formado de dos operaciones parciales: **b1**: cálculo de los valores de *refpointPos* y *refpointHpr* necesarios para definir dicha matrices (gestión de comportamiento). **b2**: generación de las matrices a partir de esos valores.
c :	Generación de la matrices *sklsrootMat* y *sklsrootsAbsMat*. Formado por dos operaciones parciales **c1**: cálculo de los valores de *sklsrootPos* y *sklsrootHpr* necesarios para definir dichas matrices (gestión de comportamiento). **c2**: generación de las matrices a partir de esos valores.
cg:	Operaciones implicadas en el procesado del culling de un nodo. Formado por dos operaciones parciales: **cg1**: Operación de recálculo de una *bsphere* a partir de la *bspheres* de sus nodos hijos. **cg2**: Comprobación de la relación entre dicha *bsphere* y el *frustum*.
dLOD:	Cálculo de distancias necesario para la gestión de nivel de detalle.

Tabla 7-1. Operaciones elementales implicadas en el procesado de los actores virtuales

7.3.1 Método de estimación del coste individual de cada operación.

Evaluar de una forma genérica el coste temporal de una determinada operación resulta muy complicado. Depende enormemente del tipo de CPU empleada, y no sólo de su frecuencia de funcionamiento, sino también, del número de ciclos de reloj que necesite para cada operación básica, de si puede o no, realizar más de una operación por ciclo de reloj, etc. El problema se puede simplificar teniendo en cuenta que el

conjunto de operaciones cuyo coste queremos estimar, consisten básicamente en la realización de sumas, multiplicaciones, divisiones y alguna raíz cuadrada, todas ellas realizadas sobre números en punto flotante.

Resulta sencillo determinar el número de sumas, multiplicaciones, divisiones y operaciones de raíz cuadrada que componen cada una de estas operaciones elementales mostradas en el apartado anterior. También es posible consultar las especificaciones de la CPU, para conocer el número de ciclos de reloj que necesita para realizar una multiplicación de 2 números flotantes, o una raíz cuadrada. Es frecuente que la capacidad de una CPU para realizar operaciones con número flotantes, aparezca expresada en *MFLOPs* (millones de operaciones en punto flotante por segundo). Sin embargo, ésta es una magnitud que no resulta adecuada en este caso, puesto que es una medida de pico, en la que no se muestra la diferencia entre los distintos tipos de operaciones, y cuyo valor solamente hace referencia al rendimiento de la CPU, desentendiéndose de características tales como el tipo de memoria empleada.

Para poder estimar de una forma precisa, el coste de las operaciones con las que estamos tratando, se va a emplear como unidad el coste real de realización una multiplicación de dos números flotantes en una determinada máquina expresado en microsegundos. Esta unidad ya fue introducida durante el estudio del coste computacional del nodo *Skeleton,* e identificada con el nombre de ***FMULT***. El coste de otras operaciones, como la suma, una división o una raíz cuadrada, podrá ser también expresado en esta unidad, calculando para ello sus costes en microsegundos, y estableciendo una relación con el coste de la multiplicación.

Para obtener el valor en microsegundos de unidad FMULT, y de los costes de las operaciones de suma, división o raíz cuadrada, se puede emplear una pequeña función que realice de forma consecutiva un número elevado de este tipo de operaciones, y mida los tiempos consumidos. El resultado de realizar este tipo de comprobación sobre un ordenador Onyx2 con una CPU R10000 a 195 MHz, es que para realizar 100.000.000 de operaciones de cada tipo son necesarios: 1.6 segundos en el caso de la multiplicación y la suma, 7.6 segundos en el caso de la división, y 10.6 segundos en el caso de la raíz cuadrada. A partir de estos resultados se puede establecer que en este tipo de máquina:

1 FMULT	-> 16 nanosegundos.
1 suma de flotantes (add)	-> 1 FMULTs
1 multiplicación de flotantes(mult)	-> 1 FMULTs
1 división de flotantes(div)	-> 5 FMULTs
1 raíz cuadrada de flotantes(sqrt)	-> 7 FMULTs

A partir de ahora, y por simplicidad, vamos a suponer que estamos trabajando en un ordenador con unas características similares, y estimar los costes de las distintas operaciones siguiendo este criterio.

7.3.2 Estimación del coste computacional de cada operación elemental.

Una vez establecida una unidad de coste temporal adecuada, obtenido el valor en unidades de tiempo dicha unidad, y expresado en *FMULTs* el coste de las operaciones de suma, división y raíz cuadrada de números flotantes, vamos a estimar el coste de las distintas operaciones elementales que intervienen en el procesado de una escena. Algunas de estas operaciones presentaran una diferencia de coste entre el caso de emplear los métodos especiales para actores, y los métodos tradicionales.

Los costes de las operaciones relacionadas con la gestión del comportamiento (**gms1**, **b1** y **c1**) no pueden ser estimados en este apartado, puesto que su valor depende totalmente del tipo de técnicas empleadas en la gestión del comportamiento de los actores.

7.3.2.1 Coste de la operación **gms2**.

El coste de generación de la matriz de transformación vinculada con un punto de articulación ha sido calculado en el apartado que trataba de los nodos *Skeleton.* La generación de dicha matriz para un punto de articulación con dos grados de libertad rotacionales, y sin tener en cuenta las características especiales de los nodos *Skeleton,* requería de la generación y multiplicación de 8 matrices de 4x4, con un coste equivalente a 908 FMULTs.

En el caso de emplear los nodos *Skeleton*, el coste computacional de era expresado según la siguiente ecuación:

coste nodo Skeleton = fTracking x fCoherencia x 29 + fTracking x 53 + fCoherencia x 19 +10

(Expresado en FMULTs, para un nodo *Skeleton* patrón con dos grados de libertad.)

Expresión según la cual, el coste varía entre 10 y 111 FMULTs dependiendo de los *factores de tracking* y *coherencia temporal*. Unos valores comunes en una aplicación de simulación podrían estar entre 20 (*fCoherencia* = 0.3 *fTracking* = 0.1) y 30 FMULTs(*fCoherencia* = 0.7 *fTracking* = 0.1).

7.3.2.2 Coste de las operaciones **b2** y **c2**.

La generación de las matrices relativa y absoluta del nodo Actor, a partir de los valores de *refpointPos*, *refpointHpr*, *sklsrootPos* y *sklsrootHpr,* habían sido evaluados en el apartado en el que se analizan los costes de procesado asociados al nodo *Actor,* identificados con los nombres de *costeSklsrootMats* (equivalente aquí a **b2**) y *costeRefpointMats* (equivalente a **c2**). Sus valores eran los siguientes:

	Procesado tradicional	Procesado Actores.
b2	454 FMULTs	fCoherencia*22 +63 FMULTs
c2	566 FMULTs	fCoherencia*85 +63 FMULTs

En una aplicación con un *fCoherencia* = 0.7, tendríamos b2 = 78 y c2 =122.

7.3.2.3 Coste de la operación **cg1**.

Para estimar el coste de recálculo de una *bounding sphere* a partir de las *bspheres* de sus hijos, se va a suponer que de media, cada nodo *Skeleton* tiene dos nodos hijos. Para calcular la esfera del nodo a partir de las otras dos, se empleará un método basado en el hecho de que su centro ha de estar sobre la línea los centros de las dos esferas iniciales, y su diámetro ha de ser el resultado de sumar la distancia que separa sus centros, más los valores de los radios de las dos esferas iniciales. De este modo, el cálculo de la nueva *bsphere* requeriría las siguientes operaciones:

- Cálculo de la distancia entre centros: 3 mults+3 sumas + 1sqrt
- Cálculo el radio de la nueva esfera: 2 sumas y una división.
- Cálculo de un vector director de la línea que une ambos centros: 3 sumas y 3 divisiones.
- Cálculo de uno de los extremos del diámetro de la nueva esfera, a partir del centro y el radio de una de las esferas y el vector director previo: 3 mults y 3 sumas.
- Cálculo del centro, a partir del punto extremo del diámetro, su radio y el vector director: 3 mults y 3 sumas.

El coste total de esta operación sería pues de 9 mults, 14 sumas, 4 divisiones y 1 sqrt, lo cual expresado en FMULTs daría un resultado de cg1 = 50 FMULTs.

7.3.2.4 Coste de la operación **cg2**.

Para realizar la comprobación de la intersección entre una *bsphere* de un nodo y el *frustum*, es necesario primero, transformar a la esfera al sistema de coordenada del *frustum*, y a continuación, calcular la distancia del centro de la esfera, a cada uno de los seis planos que componen el *frustum*.

El cálculo de la distancia de un punto a un plano es una operación muy sencilla que tan sólo requiere 3 multiplicaciones y 3 sumas. En total, la comprobación de la intersección de una *bsphere* con el *frustum* requeriría 18 mults y 18 sumas. Si a este coste se le añade el de transformar la *bsphere* al sistema de coordenadas absoluto multiplicando su centro por la matriz de transformación actual (multiplicación de matriz 4x4 por un vector = 16mults + 12 sumas), se obtiene que el coste de esta operación es de 34 mults y 30 sumas, lo cual expresado en FMULTs daría un resultado de cg2 = 64 FMULTs

7.3.2.5 Coste de la operación **dLOD**.

Para realizar el cálculo de distancia entre el punto central de un nodo LOD y el punto de vista, sería necesario realizar 3 multiplicaciones, 5 sumas y una raíz cuadrada. En realidad las distancias de cambio de LOD pueden ser representadas internamente como distancias al cuadrado. Esto evitaría tener que realizar la raíz cuadrada en el cálculo de distancias, y con ello, el cálculo de una distancia podría ser realizado con tan sólo 3 multiplicaciones y 5 sumas, es decir, dLOD: 8 FMULTs.

7.4 Estimación cualitativa de coste del procesado de una escena genérica.

En esté apartado, se va a analizar el número de operaciones elementales de cada tipo, que será necesario realizar durante el procesado de una escena, mostrándose la diferencia entre el número de operaciones que se realizaran empleando los métodos propuestos en este trabajo, y las que serían necesarias si se emplease un procesado tradicional.

Los factores que determinan el número de operaciones a realizar son los siguientes:

nActors : Número total de actores existentes en la escena.

nSkelsPerActor: Número de puntos de articulación existentes en cada actor.

fRefpointBsphsIn : Porcentaje de actores cuya *Refpoint Bsphere* está dentro del *frustum*.

fSklsrootBsphsIn : Porcentaje de actores cuya *Sklsroot Bsphere* está dentro del *frustum*.

fCheckIntBsphs: Porcentaje de actores en los cuales es adecuado realizar culling con sus *Internal Bspheres*.

fRefpointBsphsIsect: Porcentaje de actores cuya *Refpoint Bsphere* intersecta con los bordes del *frustum*.

fSklsrootBsphsIsect: Porcentaje de actores cuya *Sklsroot Bsphere* intersecta con los bordes del *frustum*.

fLODtopo: Porcentaje medio de reducción del número de nodos *Skeleton* de un actor, debido a la gestión de nivel de detalle topológico.

nBsphsPerActor: Número de *Internal Bspheres* que componen internamente al actor, incluye las *bspheres* de los nodos puntos de articulación y también de las geometrías.

Empleando dichos factores para estimar el número de operaciones de cada tipo tendríamos:

	procesado tradicional	procesado actores
gms	nActors *nSkelsPerActor	(fSklsrootBsphsIn*nActors) * (nSkelsPerActor*fLODtopo)
b	nActors	nActors
c	nActors	nActors*fRefpointBsphsIn
cg1	nActors*(1+1+nBsphsPerActor)	nActors*fCheckIntBsphs *nBsphsPerActor
cg2	nActors * (1 + fRefpointBsphsIsect + fSklsrootBsphsIsect *(nBsphsPerActor/2))	nActors * (1 + fRefpointBsphsIsect + fCheckIntBsphs *(nBsphsPerActor/2))
dLOD	nActors*(1+nSkelsPerActor)	nActors

Tabla 7-2. Estimación del número de operaciones elementales según tipo de procesado.

En la anterior tabla se puede examinar la importancia relativa de cada operación, y también, observar la forma en la el procesado específico de actores actúa reduciendo el número de operaciones a realizar en

cada fase. Esta reducción se muestra evidente teniendo en cuenta que los factores *fRefpointBsphsIn, fSklsrootBsphsIn*, *fRefpointBsphsIsect, fSklsrootBsphsIsect, fCheckIntBsphs y fLODtopo*, son valores inferiores a la unidad. En el cálculo de *cg2,* el hecho la comprobación de *culling* se realice de una forma jerárquica, ha sido representado dividiendo por 2 el valor de *nBsphsPerActor*.

En el siguiente apartado, se pretende mostrar esta mejora de una forma mucho más evidente. Para ello se sustituirán estas variables por valores concretos tomados de una escena patrón.

7.5 Estimación cuantitativa del coste de procesado de una escena patrón.

Para poder observar de una forma más clara la diferencia de coste computacional existente el procesado tradicional, y el procesado específico de actores, vamos a suponer un escenario formado por un número concreto de actores. Cada uno de estos actores presentará unas características determinadas que ayudaran a estimar de forma numérica su coste computacional.

7.5.1 Definición de la escena Patrón.

La escena patrón estará formada por 100 actores (*nActors* = 100), existiendo 30 actores cuya *Refpoint Bsphere* está total o parcialmente dentro del *frustum* (*fRefpointBsphsIn* = 0.3), 20 cuyas *Sklsroot Bsphere* está total o parcialmente dentro del frustum (*fSklsrootBsphsIn* = 0.2), 20 cuya *Refpoint Bsphere* intersecta con los bordes del frustum (*fRefpointBsphsIsect* = 0.2), 8 cuya *Sklsroot Bsphere* intersecta con los bordes del *frustum* (*fSklsrootBsphsIsect* = 0.08), y 2 actores que tienen una relación con el *frustum,* tal que es adecuado realizar comprobaciones de CULL con sus *Internal Bspheres* (*fCkeckIntBsphs* = 0.02).

Respecto a cada uno de los actores de la escena, se va a emplear un actor patrón con las siguientes características:

- Está formado por 40 nodos *Skeleton*,
- Tiene 10 nodos *Skeleton* de 3 DOFs, 20 de 2DOFs y 10 de 1 DOF. Es decir, de media 2 DOFs por nodo *Skeleton*.
- Tiene 4 niveles de detalle topológico, el primero con 40 nodos *Skeleton*, el segundo con 20 nodos *Skeleton*, el tercero con 10 y el último con ninguno. Supongamos que de media un actor se representa con la mitad de los nodos que tiene en su nivel de detalle más alto, es decir, 20 nodos *Skeleton* en este caso (fLODtopo = 0.5)
- Existen 41 nodos LOD, uno por cada nodo *Skeleton,* y uno adicional por el nodo *Actor*.
- Hay un *factor de coherencia temporal* de 0.7, es decir, de media, durante un 30% del tiempo, los valores de los DOF permanecen sin cambio con respecto al frame anterior.

7.5.2 Cálculo del número de operaciones elementales.

Si expresamos los parámetros del apartado anterior en función de los valores definidos para la escena patrón obtenemos:

nActors	= 100
nSkelsPerActor	= 40
fRefpointBsphsIn	= 0.3

fSklsrootBsphsIn = 0.2
fRefpointBsphsIsect = 0.2
fSklsrootBsphsIsect = 0.08
fCheckIntBsphs = 0.02
fLODtopo = 0.5
nBsphsPerActor = 80

Si empleamos dichos sobre las expresiones de la *Tabla 7-2* se obtiene:

	procesado tradicional	procesado actores
gms	4000	400
b	100	100
c	100	30
cg1	8200	160
cg2	460	200
dLOD	4100	100

Tabla 7-3. Número de operaciones elementales empleadas en la gestión de una escena patrón.

De la simple observación de la tabla se puede apreciar una considerable mejora entre la utilización de método de procesado especial para actores y el método tradicional.

7.5.3 División entre gestión del comportamiento y gestión del grafo de escena.

Esta mejora aún es mayor si se tienen en cuenta los métodos especiales de cálculo de *gms, b* y *c* empleados por los nodos *Actor* y *Skeleton*, así como la gestión de nivel de detalle comportamental. Para poder analizar estos aspectos hay que tener en cuenta que tanto la generación de la matriz de cada punto de articulación (gms), como la generación de las matrices del *Reference Point* y el *Skeletons Root* (b y c), están en realidad formada por dos operaciones separadas:

gms = gms1 (gestión de comportamiento) + gms2 (generación de la matriz)
b = b1 (gestión de comportamiento) + b2 (generación de la matriz)
c = c1 (gestión de comportamiento) + c2 (generación de la matriz)

De igual modo, hay que tener en cuenta que algunas de las operaciones que estamos tratando, presentan diferente coste en el caso de emplear un procesado tradicional, o el especial para actores. Así, en la tabla que sigue a continuación el coste de *gms2* es diferenciado como ***gms2t,*** si se emplea el procesado tradicional, y ***gms2a,*** en el caso de emplear el procesado de actores. Lo mismo ocurre con los valores de *c2* y *b2*, y también con los valores de *gms1*, y *c1*. La tabla se muestra dividida en dos grupos, las tres

primeras filas hacen referencia a operaciones relacionadas con la gestión del comportamiento, y el resto a operaciones propias de la gestión del grafo de escena.

	operación	procesado tradicional	procesado actores
gestión de comportamiento	gms1	4000 * gms1t	400 * gms1a
	b1	100 * b1	100 * b1
	c1	100 * c1t	30 * c1a
gestión del grafo de escena	gms2	4000 * gms2t	400 * gms2a
	b2	100 * b2t	100 * b2a
	c2	100 * c2t	30 * c2a
	cg1	8200 * cg1	190 * cg1
	cg2	460* cg2	200 * cg2
	dLOD	4100 * dLOD	100 * dLOD

Tabla 7-4. Coste asociado al procesado una escena patrón expresado en función del número de operaciones elementales.

Los costes de generación de las matrices a partir de sus valores (gms2b, b2b y c2b) se ven considerablemente reducidas por el hecho de emplear nodos *Actor* y *Skeleton* que generan matrices especializadas y aprovechan la coherencia temporal.

7.5.4 Coste de la gestión del comportamiento.

Los costes de gestión de comportamiento (gms1 y c1) se reducen significativamente por el hecho de realizar una gestión de nivel de detalle comportamental. Si definimos como ***gms1m,*** el coste medio de la gestión del comportamiento relacionado con un punto de articulación, y como ***c1m,*** el coste medio de la gestión de comportamiento relacionado con el cálculo de la posición del *Skeletons Root* del actor, y si suponemos que, por el hecho de emplear gestión de nivel de detalle comportamental, conseguimos una reducción de la gestión del comportamiento a la mitad, obtenemos las siguientes expresiones.

gms1t = gms1m, gms1a = 0.5 * gms1m, c1t = b1m y c1a = 0.5*b1m.

Si se substituyen dichas expresiones en la *Tabla 7-4*, el coste de gestión de comportamiento queda expresado como:

	operación	procesado tradicional	procesado actores
gestión del comportamiento	gms1	4000 * gms1m	200 * gms1m
	b1	100 * b1	100 * b1
	c1	100 * c1m	15 * c1m

Tabla 7-5. Coste asociado a la gestión del comportamiento de una escena patrón expresado en número de operaciones elementales.

Si, además, consideramos que las operaciones *gms1m*, *b1* y *c1m,* tienen un coste similar, obtenemos que el coste total de gestión del comportamiento puede ser expresado como:

	procesado tradicional	procesado actores
gestión del comportamiento	4200 * gms1m	315 * gms1m

Tabla 7-6. Coste total asociado a la gestión del comportamiento de una escena patrón expresado en número de operaciones gms1m.

factor de mejora en la gestión del comportamiento = 13

7.5.5 Coste de la gestión del grafo de escena.

Si los valores relacionados con la gestión del grafo de escena de la *Tabla 7-4*, son aislados en una nueva tabla, y se despejan los valores de las operaciones elementales (expresándolos ahora en FMULTs) se obtiene lo siguiente:

	operación	procesado tradicional	procesado actores
gestión del grafo de escena	gms2	4000 * 908 = 3.632.000 FMULTs	400 * gms2a FMULTs
	b2	100 * 454 = 45.400 FMULTs	100 * b2a FMULTs
	c2	100 * 566 = 56.600 FMULTs	30 * c2a FMULTs
	cg1	8200 * 50 = 410.000 FMULTs	190 * 50 = 9.500 FMULTs
	cg2	460 * 64 = 29.440 FMULTs	200 * 64 = 12.800 FMULTs
	dLOD	4100 * 8 = 32.800 FMULTs	100 * 8 = 800 FMULTs

Tabla 7-7. Coste asociado al procesado del grafo de escena una escena patrón.

El coste total de la gestión del grafo de escena según el procesado tradicional sería

costeGrafoTradicional = 4.206.240 FMULTs

Coste gestión del grafo de escena mediante Procesado Tradicional

El coste total de la gestión del grafo de escena, siguiendo los métodos propuestos en este trabajo sería el indicado por la expresión:

coste gestión grafo procesado actores = 400 *gms2a + 100*b2a + 30*c2a + 23100

Los valores de gms2a (coste de generación de la matrices de un nodo *Skeleton*), b2a y c2a (costes de generación de las matrices del *Refpoint* y el *Sklsroot* de un nodo *Actor*) han sido calculadas en los apartados en los que se trata del coste computacional de los nodos Actor y Skeleton, y expresados como:

gms2a = fTfracking*fCoherencia*29 + fTracking*53 + fCoherencia*19 +10.
b2a = fCoherencia*22 +63
c2a = fCoherencia*85 +63

Despejando estos valores en la expresión anterior se obtiene:

costeGrafoActores = 11600*fTrack*fCohe +21200*fTrack+12350*fCohe + 35290 FMULTs

Coste gestión del grafo de escena mediante Procesado de Actores.

Si esta expresión es representada de forma gráfica se obtiene:

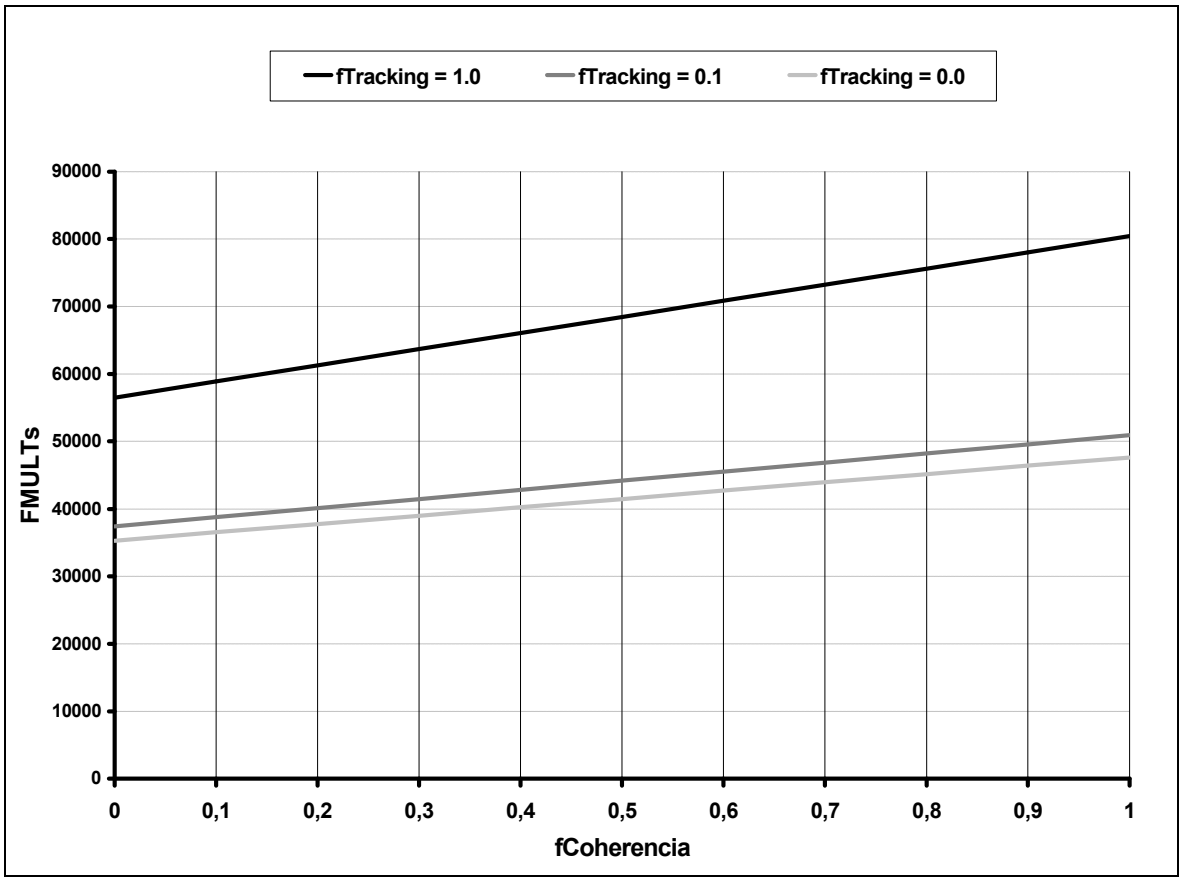


Figura 7-1. Coste de procesado del grafo de escena de una escena patrón en función del *factor de coherencia temporal* y el *factor de tracking*.

Los valores obtenidos varían entre 35290 FMULTs (*fTracking* = 0.0; *fCoherencia* = 0.0), y 80440 FMULTs (*fTracking* = 1.0; *fCoherencia* = 1.0). Es decir, es entre 55 y de 120 veces más rápido que el procesado realizado mediante un grado de escena tradicional.

El factor de mejora en la gestión del grafo de escena varía en función de los valores de *fTracking* y *fCoherencia*.

Factor de mejora en la gestión del grafo de escena = 55< x <120.

7.5.6 Coste global y factor de mejora.

El factor de mejora global es calculado en función de los costes de procesado del grafo de escena, y la gestión del comportamiento. El factor de mejora global mínimo se va a dar cuando el coste de la gestión del comportamiento sea elevado, y éste será máximo, cuando la gestión de comportamiento tenga un coste temporal muy reducido.

El coste global de la gestión de los actores de la escena patrón, según los métodos de un grafo de escena tradicional puede ser expresado como:

costeGlobalProcTradicional = 4.206.240 + 4200 x gms1m FMULTs

Y el coste de gestión de los actores, según los métodos propuestos en este trabajo puede ser expresado como

costeGlobalProcActores = costeGrafoActores +315 x gms1m FMULTs

El factor de mejora global puede ser expresado en función del coste en FMULTs asociado a la operación de *gms1m,* el *factor de coherencia* y el *factor de tracking* mediante la siguiente ecuación:

$$factorMejoraGlobal = \frac{\mathbf{4.206.240} + \mathbf{4200} \times gms\mathbf{1}m}{\cos teGrafoActores + \mathbf{315} \times gms\mathbf{1}m}$$

Si dicha expresión es representada de forma gráfica se obtiene:

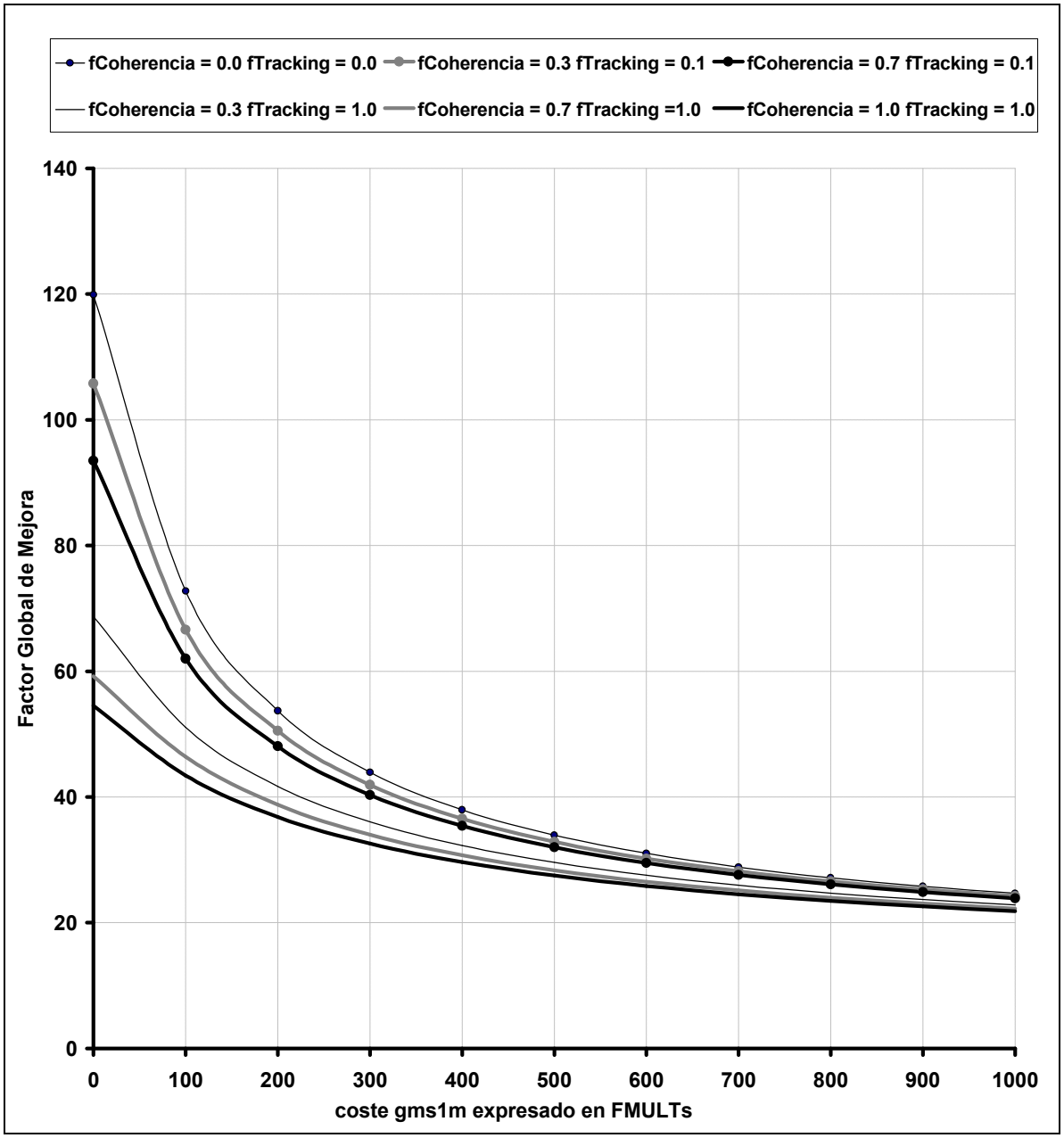


Figura 7-2. Factor de mejora global, en función del coste medio de gestión del comportamiento de un nodo *Skeleton,* y los *factores de Coherencia* y *Tracking,*

En esta representación se puede observar como se puede conseguir un factor de mejora entre 13 (valores de *gms1m* muy grandes) y 125 (valores de *gms1m* pequeños, y *factores de tracking* y *coherencia* cercanos a 0).

En la práctica será habitual trabajar con valores de *gms1m* menores de 50 FMULTs, y también con *factores de coherencia temporal* cercanos a 0.7 y *factores de tracking* cercanos a 0.1. Si consideramos, además, que la escena patrón va a ser simulada en un ordenador Onyx2 con una CPU R10000 a 195 Mhz como el citado en el apartado 7.3.1 en el cual el valor de FMULT era de 16 nanosegundos.

fTracking	= 0.1	gms1m	= 50 FMULTs
fCoherencia	= 0.7	1 FMULT	= 16 nanosegundos.

costeProcesadoTradicional = (4.206.240 + 4.200*50)* 16 nanoSecs = 70,6 miliSegundos.

costeProcesadoActores = (46867 + 314*50)*16 nanoSecs = 1,001 miliSegundos

En una escena de simulación a 50 frames por segundo, disponemos de 20 milisegundos para el dibujado de cada frame. Si suponemos que tan sólo la mitad de ese tiempo puede ser destinado al procesado de los actores virtuales puesto la CPU ha de dedicarse a otras tareas tales como el la gestión del resto de los elementos que no son actores virtuales, gestión de periféricos de entrada y salida, comunicación con la tarjeta gráfica etc., contaremos con tan sólo 10 milisegundos para procesar todos los actores de la escena.

En el caso del procesado tradicional, el coste es 7 veces superior a este valor, con lo cual la simulación nunca podría funcionar en una escena de simulación de estos requisitos. Pudiéndose como mucho representar esta escena con un frame-rate de 6 o 7 imágenes por segundo.

En el caso del procesado específico de actores, el coste es 10 veces inferior, con lo cual no solamente podría ser representada sin ningún problema, sino que incluso sería posible incrementar la cantidad de actores a simular en un factor de 10.

7.6 Conclusiones.

En este capítulo se han detallado cuales son las operaciones elementales que se realizan durante el procesado de una escena con varios actores virtuales. Ello ha permitido mostrar la importancia relativa de cada operación, y también poder realizar comparaciones entre el número de cada tipo de operación que sería necesario realizar en el caso de un procesado tradicional, y en el caso del procesado específico propuesto en este trabajo.

Para poder afrontar la complejidad de la estimación del coste total de este procesado, se ha propuesto la utilización de una escena patrón formada por 100 actores de una características similares a las necesarias en una situación real. En esta escena patrón, se ha estudiado por separado el coste de la gestión del comportamiento, y el coste de la gestión del grafo de escena. El resultado ha sido la obtención de dos expresiones algebraicas. Mediante la primera de ellas se puede calcular el coste de la gestión del comportamiento, en función del coste medio de la operación que define los valores de los grados de libertad de un nodo *Skeleton.* Mediante la segunda, se puede calcular el coste de la gestión del grafo de escena, en función de los *factores de Coherencia* y*Tracking*. Ambas expresiones han sido combinadas para poder conocer el coste global del procesado de los actores, y también el factor de mejora existente entre los métodos de gestión propuestos en este trabajo y los métodos tradicionales.

El resultado final ha sido una expresión gráfica en la que se muestra que para valores comunes de gestión del comportamiento (gms1m < 100 FMULTs), el factor de mejora se mueve entre 50 y 120, y en el peor de los casos, para gestiones de comportamiento extrañamente altas, el factor de mejora es de 13.

Por último, se han empleado estas expresiones para conocer que ocurriría si la escena patrón fuese ejecutada en un ordenador Onyx 2 con una CPU R10000 de 195 MHz, obteniéndose como resultado, que, en el caso tradicional se obtendría como máximo un frame-rate de 6 o 7 imágenes por segundo, y en el caso del procesado propuesto en este trabajo se podría alcanzar un frame-rate de 50 imágenes por segundo con una cantidad de actores cercana a 1000.

PARTE IV
IMPLEMENTACIÓN

Capítulo 8. Implementación práctica.

8.1 Índice.

8.2 Introducción.

Las estructuras de datos y métodos de control descritos en esta Tesis Doctoral, han sido empleados para implementar un prototipo de una librería. Esta librería, que presenta una interfaz de programación en ANSI C, permite la creación de aplicaciones de simulación que requieran la presencia de actores virtuales. El objetivo principal de la implementación de este prototipo es demostrar la capacidad de obtener un aprovechamiento práctico de los resultados de este trabajo, así como mostrar que las estructuras y datos presentados se comportan de la forma descrita en los capítulos teóricos.

Este prototipo de librería será empleada en el capítulo siguiente para ampliar una aplicación de simulación ya existente (la visualización de la database *"Performer Town"* sobre aplicación *"perfly"* desarrollada con *OpenGl Performer*), de tal modo que incorpore una gran cantidad de actores realizando distintos tipos de actividades.

En este capítulo se detallarán los siguientes aspectos, relacionados con la implementación de la librería de actores virtuales, y la integración de actores virtuales en aplicaciones de simulación reales:

- Metodología de Ingeniería de Software empleada para la generación de la librería.

- Descripción del API del prototipo de librería implementado. Esta librería contiene internamente la definición de las clases principales descritas en esta Tesis Doctoral, y proporciona un conjunto de funciones que dan acceso a dichas clases.

- Descripción del método que permite que una misma aplicación con actores virtuales pueda ser integrada en diferentes grafos de escena.

- Definición de una arquitectura por capas que permite afrontar el problema de la creación de una aplicación con actores virtuales de una forma modular, facilitando entre otras cosas, la reutilización de los módulos, la independencia de la plataforma y del grafo de escena, y la integración del trabajo de diferentes desarrolladores.

- Descripción de los pasos necesarios para el desarrollo de una aplicación con actores virtuales mediante el empleo de la librería, y la arquitectura propuestas.

8.3 Metodología de ingeniería del software.

Si bien éste es el primer apartado en el que se trata de ingeniería del software, se puede observar que la gran parte de este trabajo de tesis ha estado centrada en la obtención de un modelo de objetos adecuado. La elección de una metodología de ingeniería del software orientada a objetos es adecuada por dos razones principales: Por un lado se trata de definir estructuras y métodos que han de ser añadidos a un grafo de escena, y es norma común que todos los grafos de escena presenten orientación a objetos. Por otro lado, la técnica de orientación a objetos resulta, en general, ideal para la implementación de cualquier tipo de sistema que intente simular el mundo real.

El prototipo de librería desarrollado proporciona un API formado por 134 funciones. Toda la librería se organiza en torno a un número muy reducido de estructuras de datos principales, siendo la misión principal del API proporcionar acceso a estas estructuras.

Durante el ciclo de desarrollo de esta librería se han aplicado conceptos orientados a objetos tanto en las fases de análisis, como en la fase de diseño e implementación. Las clases empleadas en las tres fases han sido básicamente las mismas, con la excepción de algunas clases auxiliares que ha sido introducidas en las fases de diseño e implementación. De las tres clases de modelos empleados por la metodología OMT para describir sistemas, la que ha prestado más utilidad en la implementación de esta librería, ha sido el modelo de objetos, siendo mínimos los modelos dinámico y funcional.

En los siguientes apartados se presenta una definición del problema desde el punto de vista de la orientación a objetos, y a continuación, se describen los aspectos más interesantes de las fases de análisis, implementación, y diseño.

8.3.1 Definición del problema.

Se pretende desarrollar un sistema que permita incorporar actores virtuales en entorno de simulación ya existente. El problema principal es que la implementación de una aplicación empleando los sistemas tradicionales tiene un coste computacional muy elevado, y no existe ningún método estándar para la definición de actores que permita reutilizar código, e integrar el trabajo de distintos desarrolladores.

Para determinar los requisitos funcionales de la librería se ha analizado el estado del arte de las investigaciones relacionadas con actores virtuales, las características de las aplicaciones actuales que los emplean, y también las características de las librerías que permiten el diseño de aplicaciones de simulación. La situación actual de las investigaciones relacionadas con actores virtuales está caracterizada por la preocupación por los aspectos de más alto nivel. En relación con las aplicaciones, éstas suelen consistir desarrollos verticales en los que se ha implementado algún actor a partir de nodos tradicionales de un grafo de escena, o directamente a partir de una librería gráfica de bajo nivel, presentando en muchos casos problemas de eficiencia y de integración con aplicaciones estándar.

Del análisis del estado del arte de la simulación, se desprende que la utilización de grafos de escena ha sido claramente aceptado como método estándar para la implementación de aplicaciones de simulación. Esto hace que la librería haya de estar pensada para trabajar de forma conjunta con los grafos de escena. En la actualidad existen distintos tipos de grafos de escena, pero se ha hecho un análisis de sus características comunes, buscando que la librería sea capaz de funcionar sobre cualquier grafo de escena que cumpla unos requisitos mínimos.

En el proceso total de definición de los actores virtuales, la librería ha de centrarse especialmente en los aspectos relacionados con la integración con el grafo de escena, la estandarización, el control de alto nivel y el rendimiento. La librería ha de actuar a dos niveles: proporcionando una estructura articulada totalmente integrada en el grafo de escena, y que, además, sea muy eficiente, estándar y sencilla de controlar, y también proporcionando un buen soporte para la definición de módulos de alto nivel.

Por lo general, el usuario quiere disponer de actores que realicen actividades complejas en un determinado entorno de simulación, pero en muchos casos los actores virtuales no son más que uno de los aspectos de la simulación. El desarrollador de aplicación no desea invertir mucho tiempo en la incorporación de los actores virtuales, siendo su deseo controlar las acciones de los actores mediante comandos de muy alto nivel, tales como "muévete hasta llegar a ese punto", o "siéntate en ese lugar", o incluso disponer de actores inteligentes, capaces de analizar el entorno en el que se mueven, y actuar de forma autónoma. La cantidad de trabajo necesario para conseguir ese objetivo suele ser en general bastante elevado. Es norma común que como base se emplee de un grafo de escena o directamente una librería gráfica de más bajo nivel como la OpenGl, y se implementen todas las capas necesarias hasta conseguir el objetivo final, es también habitual que el desarrollador no desee preocuparse de aspectos de bajo nivel tales como la definición y gestión a bajo nivel de la estructura articulada del actor. También sería un objetivo deseable, lograr que las capas de alto nivel diseñadas por un desarrollador en un determinado entorno, pudiesen ser empleadas por otro desarrollador en otro entorno diferente.

8.3.1.1 Requisitos funcionales.

Los requisitos funcionales de esta librería pueden ser clasificados en cinco grandes grupos: interacción con el grafo de escena, estandarización, rendimiento, extensibilidad y gestión de la complejidad. Veamos cada uno por separado:

1.Desde el punto de vista de interacción con el grafo de escena.

La librería ha de estar ideada para ser empleada sobre aplicación de simulación ya existente. Para ello ha de adaptarse a las características de un grafo de escena genérico:

- Adaptación al estado del arte en el mundo de simulación visual en tiempo real. Utilización como base de la estructura de grafo de escena.
- El grafo de escena proporciona su propio método de culling. Los actores han de ser afectados por el método de culling estandar de la escena.
- El grafo de escena proporciona un método de gestión de nivel de detalle. Los actores virtuales han de ser afectados por el método de gestión de nivel de detalle.
- Los actores dentro de la escena no han de actuar como elementos aislados. Un actor puede emplear un objeto de la escena (por ejemplo una herramienta), y un objeto de la escena puede contener a uno o varios actores (por ejemplo un vehículo).

2.Desde el punto de vista de la estandarización.

Es necesario un método estándar que permita la definición de la estructura articulada de un actor virtual, este método ha de ser suficientemente genérico como para poder definir cualquier tipo de actor, y, además, ha de resultar sencillo, robusto y flexible. Adionalmente, ha de ser independiente de la plataforma, del sistema operativo, e incluso del grafo de escena, de ese modo los desarrollos pueden ser realizados de una forma que asegure su reutilización.

3.Desde el punto de vista del rendimiento.

Una aplicación de simulación compleja, puede requerir una cantidad muy elevada de actores que han de ser totalmente actualizados y dibujados 50 veces por segundo, además, es habitual que la gestión de actores sea solamente uno de las tareas que consume tiempo de cálculo. Se hace necesario un análisis de cuales son los puntos críticos en la gestión de actores virtuales, y realizar una implementación que optimice el rendimiento.

4.Desde el punto de vista de la extensibilidad.

La necesidad de extensibilidad también constituye un requisito de la estandarización. La librería centra su atención en gestión de los aspectos de nivel de nivel bajo y medio de los actores virtuales, las capas relacionadas los modelos motor y comportamental han de ser implementadas mediante módulos externos de más alto nivel. La librería de gestión de actores virtuales ha de facilitar al máximo la creación e integración de estos módulos.

5.Desde el punto de vista de gestión de la complejidad.

El hecho de que una aplicación de simulación pueda tener más de 1000 actores desarrollando distintos tipos de actividades, no solamente afecta al rendimiento, también puede hacer que se presenten problemas en la propia gestión de la aplicación. Estos son algunos de los requisitos necesarios para reducir la complejidad en gestión de proyectos grandes:

- Necesidad de crear una representación sencilla de actor virtuales en el grafo de escena.

- Necesidad de crear una interfaz directa entre el usuario y el actor virtual, que oculte la complejidad jerárquica del actor.
- Necesidad de algún tipo de estructura que almacene la información relacionada con el comportamiento del actor, así como informaciones que serían comunes a varios actores de la misma especie.
- Capacidad de expansión y trabajo modular y multicapa.
- Posibilidad de borrar y crear actores de una forma rápida para la realización de simulaciones macroscópicas.

8.3.2 Análisis.

8.3.2.1 Modelo de objetos.

El modelo objeto esta formado por un total de 5 módulos, organizados en 2 módulos principales:

- Módulo de incorporación de actores virtuales al grafo de escena.
- Módulo de gestión de informaciones genéricas y de alto nivel.

Y 3 Módulos secundarios:

- Módulo de soporte para extensiones.
- Módulo para independencia del grafo de escena.
- Módulo para la gestión global de los actores en la escena.

Módulo de incorporación de actores virtuales en el grafo de escena.

Este módulo se preocupa de ampliar el funcionamiento de los grafos de escena actuales, de tal modo que puedan ser empleados de una forma adecuada en la representación de actores virtuales. Centra su atención en la estandarización, la eficiencia computacional, y también a los métodos de gestión de culling y nivel de detalle. Esta formado por tres clases principales: *vaSkeleton*, *vaActor* y *vaLod*, cada una de las cuales se comporta como nodo del grafo de escena:

- **Clase *vaSkeleton*:** Hace referencia a un nodo que define un punto de articulación de una estructura articulada (por ejemplo la rodilla, o el codo de un actor virtual). Es la clase más importante, ha de ser implementada de modo sea suficientemente flexible como para poder definir cualquier tipo de estructura articulada. Ha presentar un coste computacional muy bajo.
- **Clase *vaActor:*** Define el nodo que actúa como padre del subgrafo de escena que representa a un actor virtual. Actúa como centro de control de todos los nodos *vaSkeleton* que lo componen.
- **Clase *vaLod*:** Esta clase es mucho menos importante que las dos anteriores. Básicamente es un nodo mediante el cual se pueden controlar el nivel de detalle con el que se representan las distintas geometrías que componen al actor virtual.

Para completar el modelo de objetos de este módulo, es necesario tener en cuenta una clase adicional que represente a un nodo grupo genérico. Esta clase es proporcionada por el grafo de escena, y las otras tres

clases heredan de ella las propiedades que les permiten estar conectadas con el resto de los nodos del grafo de escena.

Módulo de gestión de informaciones genéricas y de alto nivel.

La misión principal de esta librería es incorporar al grafo de escena tradicional, un conjunto reducido de nodos que permitan implementar actores virtuales de una forma eficiente, estándar y sencilla. Para ello, no es suficiente con la implementación de nuevos tipos de nodos: existen muchas informaciones que son de demasiado alto nivel para ser almacenadas en los nodos del grafo de escena, así como informaciones que son comunes a varios actores. Estas informacines han de ser almacenadas en un lugar diferente al grafo de escena. La clase *vaActorclass* está preparada para almacenar las informaciones genéricas y de alto nivel compartidas por una determinada especie de actores. Esta clase emplea internamente otras clases menos importantes, que almacenan la información de alto nivel correspondiente con un punto de articulación (*vaSkelprot*), y también la información de alto nivel relacionada con un grado de libertad (*vaDof*):

- **Clase *vaActclass*:** Es la clase principal encargada de almacenar todas las informaciones genéricas y de alto nivel compartidas por un grupo de actores de una misma especie. Un actor virtual pertenecerá a una determinada "clase de actores" (*vaActclass*), por ejemplo, todos los actores de tipo humano existentes en una aplicación, pueden pertenecer a una misma clase de actor de nombre *"human"*. Una *vaActclass* contiene una lista con todos sus actores. Una aplicación puede emplear varias *vaActclass* diferentes .
- **Clase *vaSkelprot*:** Define las propiedades de alto nivel de un punto de articulación de un actor, los objetos de esta clase se pueden conectar entre sí para poder representar la estructura articulada del actor.
- **Clase *vaDof:*** Define las características de un grado de libertad. Actúa como clase auxiliar junto con la clase *vaSkelprot* para la definición de la estructura articulada de la clase de actor. Si un punto de articulación dispone de tres grados de libertad, éste será representado como un objeto de tipo *vaSkelprot* referenciando a tres objetos de tipo *vaDof.*

Módulo de soporte para extensiones.

Una de las necesidades principales de la integración de actores virtuales en una aplicación de simulación, es la definición de código que describa su comportamiento. El comportamiento de los actores constituye una capa de alto nivel que no es objetivo de esta librería, sin embargo, proporciona un método que permite encapsular cada pequeño bloque de gestión de comportamiento, de tal modo que su integración en la aplicación sea sencilla, estándar, y sobre todo, reutilizable. La **Clase *vaExtension*** es la principal encargada de implementar este soporte modular de las informaciones de alto nivel. Una extensión (por ejemplo un módulo encargado del control de la dirección mirada de un actor), puede ser empleada por distintas clases de actores. Si una clase de actor ha sido vinculada con una determinada extensión, todos sus actores disponen de esa capacidad. Cuando una extensión es asociada una clase de actor, suele ser necesario algún tipo de configuración. Las clases *vaActclass*, *vaSkelprot*, *vaDof* y *vaActor* tienen zonas

preparadas para almacenar toda la informaciones particulares relativas al uso de los módulos de extensión.

Módulo para la independencia del grafo de escena.

Este módulo se encarga de hacer que la librería de actores sea integrada sobre cualquier grafo de escena que cumpla unos requisitos mínimos. Mediante este módulo, se da a conocer a la librería de actores virtuales, los métodos para crear un nodo grupo, para asociar datos de usuario o funciones de callback a un nodo, para conocer la hora del sistema, o las características de la cámara desde la que se está viendo la escena. El soporte a la independencia del grafo de escena está encapsulado en la **Clase *vaSgcfg*** (abreviatura de "Scene Graph Configurer").

Módulo para la gestión global de los actores en la escena.

Está formado por la **Clase *vaManager*** que controla aspectos como la inicialización de la librería, la gestión del número de Extensiones, o el número de Clases de Actores a emplear en la simulación. Y también, por la **Clase *vaStats***, encargada de proporcionar distintos tipo de informaciones a cerca del rendimiento de la simulación.

8.3.2.2 Modelo dinámico y funcional.

Los modelos dinámico y funcional, pueden ser muy importantes en la definición de los modelos motor y comportamental de los actores virtuales. Sin embargo, aunque la librería proporciona soporte para la creación de estos modelos, se trata de una librería de gestión de aspectos de medio y bajo nivel, caracterizada por la importancia del modelo de objetos, frente a unos modelos dinámico y funcional poco significativos.

En este contexto, los modelos dinámico y funcional vienen establecidos por el proceso de recorrido del grafo de escena, y se puede resumir en los siguiente secuencia:

1 Se actualizan los *Reference Points* de todos los actores de la escena.
2 Se lanza el proceso de recorrido del grafo de escena.

Para cada actor:

3 Se determina si su *Refpoint Bsphere* está dentro del frustum, en tal caso llama a las funciones de comportamiento que actualizan la posición del *Skeletons Root*, en caso contrario, el actor resulta descartado.
4 Se determina si su *Sklsroot Bsphere* está dentro del frustum, en caso contrario el actor resulta descartado.
5 Se calcula su distancia a la cámara, y se llaman a los módulos de gestión del comportamiento de las extensiones, y también al propio del actor. Como resultado final se actualiza el estado interno del actor, y refrescan los valores de sus grados de libertad. El número de grados de libertad modificados dependerá de la distancia del actor a la cámara (nivel de detalle topológico).

6 Para cada nodo *vaSkeleton* del actor, se determina si alguno de sus grados de libertad ha sido modificado, y en tal caso, se recalcula su matriz de transformación.

8.3.3 Diseño.

Como ya se ha citado, la arquitectura de la librería está caracterizada por el predominio del modelo de objetos con respecto al modelo dinámico y funcional, encajando en el prototipo de entorno de arquitectura conocido como Simulación Dinámica **[RUM96],** encargado de la simulación de objetos del mundo real en evolución. La arquitectura del sistema debe ser contemplada desde dos ópticas diferentes:

- La librería de actores como subsistema dentro un sistema global de particiones que define una aplicación de simulación.
- La librería de actores como un sistema complejo compuesto de varios subsistemas internos organizados en forma de capas de nivel bajo y medio.

Teniendo en cuenta esta doble óptica, el objetivo final del diseño de la librería ha de ser complementado con la propuesta de una arquitectura que contemple todos los aspectos de la integración de los actores en la aplicación de simulación. Esta arquitectura es mostrada en detalle en el *Apartado 8.6*, y consiste básicamente en la definición de un API de muy alto nivel que permita gestionar cómodamente a cada tipo de actor, y una capa superior que indique que actividades tienen que desarrollar los actores empleando para ello los APIs de alto nivel definidos para cada tipo de actor.

La librería de actores está, además, constituida internamente por diferentes subsistemas, coincidentes con los módulos descritos en el análisis.

8.3.4 Implementación.

Para la implementación de la librería se ha empleado lenguaje C, pero aplicando el paradigma de orientación a objetos. La elección de un lenguaje no orientado a objetos para la implementación de un diseño orientado a objetos puede parecer no adecuada, pero debido a las características especiales de este trabajo (no necesidad de mecanismos de herencia ni polimorfismo), y el hecho de que tenga como intención integrarse en cualquier tipo de grafo de escena, se ha optado por almacenar los atributos de cada clase dentro de un *struct* del lenguaje C (todas estas estructuras han sido detalladas en distintos puntos de esta memoria de tesis).

La generación de un adecuado modelo de objetos durante la fase de análisis resulto de gran utilidad en las fases de diseño e implementación de las clases *vaActor*, *vaSkeleton* y *vaActclass*. Las fases de diseño e implementación pusieron de manifiesto algunos defectos del análisis en relación con la clase *vaExtension* que necesito ser redefinida hasta adoptar el aspecto actual, de igual modo también necesito ser ajustada la clase encargada de proporcionar independencia del grado de escena (*vaSgcfg*). La necesidad de clases que forman parte del módulo de gestión global de los actores en la escena (*vaManager* y *vaStats*), resulto evidente durante la fase de implementación

La librería desarrollada está formada por un total de 10 módulos, cada uno de ellos implementando una de las clases descritas anteriormente. El resultado final ha sido una librería con un API de 134 funciones, definido mediante dos ficheros de cabecera: el fichero *"vaApi.h"* (119 funciones), encargado de proporcionar acceso a las clases principales, y el fichero *"vaSgcfg.h"* (15 funciones) que proporciona acceso a las funciones encargadas de configurar a la librería para trabajar sobre un grafo de escena en particular.

Para comprobar la fiabilidad del software se ha realizado un ejemplo de uso de la librería suficientemente significativo. Dicho ejemplo consiste en la integración de un gran número de actores virtuales (superior al millar) sobre una escena de simulación clásica. En dicho ejemplo se emplea de forma exhaustiva las funciones proporcionadas por la librería de gestión de actores, constituyendo, además, una simulación suficientemente compleja como para apreciar los beneficios de la utilización de la arquitectura modular propuesta. Todos los aspectos del ejemplo de aplicación son mostrados en detalle en el capítulo siguiente.

Respecto a la importancia relativa de cada fase del desarrollo de la librería, la mayor cantidad de tiempo, aproximadamente un 60%, fue empleada en el Análisis, un 30% en las fases de Diseño e Implementación y un 10% en el desarrollo del ejemplo de aplicación. En relación con la importancia relativa de los módulos descritos en el modelo objeto, la mayor parte del tiempo se empleo en la definición del módulo de incorporación de actores virtuales en la escena (40%), seguido del módulo de gestión de informaciones genéricas y de alto nivel (30 %), el módulo de soporte para extensiones (20%), el módulo para la independencia del grafo de escena (15%), y por último el módulo de gestión global de actores en la escena (5%).

La eficiencia de las estructuras y métodos propuestos en este trabajo ha sido demostrada de forma teórica en el capítulo anterior. El ejemplo de aplicación que será presentado en el capítulo siguiente servirá para demostrar la correspondencia entre la eficiencia estimada en la teoría y los resultados obtenidos en la práctica.

Respecto a la generalidad de la librería, su modularidad, su preocupación por la estandarización, su independencia del grafo de escena y su capacidad de extensión permite que todos los desarrollos realizados mediante esta librería sean fácilmente integrables y reutilizables. Un desarrollador de una aplicación de simulación puede que necesite únicamente preocuparse de la integración de unos actores ya existentes en la aplicación final, o del desarrollo de algún módulo de comportamiento. En el caso de que necesite definir una clase de actor completa, ésta puede ser desarollada de tal modo que pueda ser reutilizada en algún proyecto posterior, o ampliada con algún módulo que le permita realizar algún tipo de actividad diferente.

8.4 Librería para la definición y manejo en tiempo real de actores virtuales.

Todas las estructuras y métodos definidos en esta tesis han sido incorporados dentro de una librería desarrollada en ANSI C. Mediante esta librería es posible definir los actores virtuales que puedan ser necesarios en una escena de simulación, y mediante el mecanismo de extensión previsto por la librería, es posible añadir los módulos de gestión del comportamiento que sean necesarios. La librería proporciona un API de 134 funciones cuyos prototipos aparecen en dos ficheros diferentes: el fichero **"vaApi.h"** (119 funciones), que contiene los prototipos de las funciones que dan acceso a las estructuras de datos principales (*vaActclass*, *vaActor*, *vaSkeleton*, etc.), y el fichero **"vaSgCfg.h"** (15 funciones), que contiene los prototipos de las funciones de configuración de la librería de actores virtuales para su trabajo sobre un grafo de escena en particular. Todos los comandos de la librería de actores virtuales comienzan con el prefijo "va".

El contenido de este apartado consistirá en una descripción directa del contenido del fichero *"vaApi.h"*. El fichero aparece fragmentado en varios trozos con el fin de poder intercalar comentarios sobre la forma su forma de utilización.

```
#ifndef __VA_API_H__
#define __VA_API_H__

//---- Tipos de estructuras y funciones principales.--------------------------//
typedef struct vaActclassStruct         vaActclass;
typedef struct vaSkelprotStruct         vaSkelprot;
typedef struct vaDofStruct              vaDof;
typedef struct vaExtensionStruct        vaExtension;
typedef struct vaActorStruct            vaActor;
typedef struct vaLodStruct              vaLod;

typedef void * (*vaActorBehaDataCreateFunc)(vaActor *act);
typedef void   (*vaActorBehaDataDeleteFunc)(void *actBehaData);
typedef void   (*vaActorBehaFunc)(      vaActor *act);
typedef void   (*vaActorDrawFunc)(      vaActor *act);

//---- Tipos y funciones auxiliares.-----------------------------------------//
typedef float float3[3];
typedef float float4[4];
typedef float float16[16];

extern void  float3Copy( float3 dst,   float3 orig);
extern void  float4Copy( float4 dst,   float4 orig);
extern void  float16Copy(float16 dst, float16 orig);
extern float angleLinearGoto(float dest, float orig, float velo, float t);
```

Las estructuras ***vaActclass***, ***vaSkelprot***, ***vaDof***, ***vaExtension***, ***vaActor*** y ***vaLod***, son las descritas en apartados anteriores de este trabajo, y son empleadas directamente por el API de gestión de actores virtuales. Como se puede observar no aparece la estructura *vaSkeleton*, esto es debido a que todos los accesos al actor virtual se realizan de forma indirecta a través de la estructura *vaActor*.

Las funciones ***vaActorBehaDataCreateFunc***, ***vaActorBehaDataDeleteFunc***, ***vaActorInitFunc*** *y* ***vaActorBehaFunc*** son empleadas como funciones de callback, su forma de empleo es descrita más adelante.

Los tipos ***float3***, ***float4*** y ***float16*** son empleados para poder manejar de forma más cómoda posiciones u orientaciones en 3D (el significado de un *float3* puede ser entendido como las coordenadas x, y, z de un punto, o también como una orientación codificada en heading, pitch y roll); Almacenar la información de una *bounding sphere* (el contenido de un *float4* puede ser empleado para almacenar el centro de la esfera en los tres primeros valores, y su radio en el cuarto), o la ecuación de un plano; Y una matriz de transformación (un *float16* se emplea para almacenar los valores de una matriz de transformación 4x4). Las funciones ***float3Copy()***, ***float4Copy()*** y ***float16Copy()*** sirven para realizar copias entre este tipo de elementos, y la función ***angleLinearGoto()*** se emplea para realizar una transición suave entre dos valores angulares.

```
//---- Funciones de control global ----------------------------------------//
extern void   vaInit();
extern void   vaEnd();

extern void   vaSetEyePosition(float3 pos);
extern void   vaGetEyePosition(float3 pos);

extern void   vaSetViewFrustum(float pNear,   float pFar, float pLeft, float pRight, float pBottom, float pTop);
extern void   vaGetViewFrustum(float *pNear, float *pFar, float *pLeft, float *pRight, float *pBottom, float *pTop);

extern void   vaSetLodScale(float scale);
extern float  vaGetLodScale();

extern void   vaUpdateSim();

extern float  vaGetCurDeltaTime();
extern float  vaGetCurTime();

extern int                vaAddActclass( vaActclass *ac);
extern int                vaGetActclassNumber();
extern vaActclass *   vaGetActclass( int index);

extern int                vaAddExtension(  vaExtension *ext);
extern int                vaGetExtensionNumber();
extern vaExtension * vaGetExtension( int index);
```

Las funciones ***vaInit()*** y ***vaEnd()*** sirven para inicializar la librería de gestión de actores virtuales, y también para liberar toda la memoria ocupada durante su utilización. Han de ser llamadas al principio y al final de la aplicación.

Las funciones ***vaSetEyePosition()*** y ***vaGetEyePosition()*** sirven para fijar y consultar la posición de la cámara desde la que se está visualizando la escena. La posición de la cámara en la escena es controlada por la aplicación, siendo responsabilidad de ésta informar a la librería de actores, cada vez que se produzca un cambio, de este modo es posible gestionar de forma adecuada algunos aspectos como los cambios de nivel de detalle.

Las funciones ***vaSetViewFrustum()*** y ***vaGetViewFrustum()*** se emplean para fijar y consultar los valores que definen el *frustum.* Estos valores son empleados internamente para definir 6 planos con los cuales se analizará la relación entre las *bounding spheres* implicadas en el *culling* de los actores virtuales, y la pirámide de visión.

De una forma similar, mediante las funciones ***vaSetLodScale()*** y ***vaGetLodScale()***, se puede fijar y consultar un valor que modifica las distancias de cambio de nivel de detalle.

La función ***vaUpdateSim()*** ha de ser llamada una vez por cada frame, dispara la actualización del estado interno de todos los actores virtuales existentes en la escena.

Las funciones ***vaGetCurTime()*** y ***vaGetCurDeltaTime()***, retornan respectivamente, el tiempo transcurrido (en segundos), desde el inicio de la aplicación, o desde el frame anterior.

Las funciones ***vaAddActclass()***, ***vaGetActclassNumber()*** y ***vaGetActclass()*** permiten trabajar con la lista total de clases de actores gestionada por la librería de actores virtuales. La primera función permite añadir una nueva clase de actores, la segunda consultar el número total de clases de actores gestionado por la librería, y la última obtener un puntero a una *vaActclass* a través de su índice. Algo similar ocurre con las funciones ***vaAddExtension()***, ***vaGetExtensionNumber()*** y ***vaGetExtension().***

```
//==== ACTORCLASS ============================================================//

//---- vaDof ---------------------------------------------------------------//
extern vaDof *vaDofNew(      char *name);
extern void    vaDofDelete(       vaDof *dof);
extern void    vaDofSetMinMax(  vaDof *dof, float  min,  float  max);
extern void    vaDofSetRanges(  vaDof *dof, float  rMin, float  rMax, float  def);
extern int      vaDofGetIndex(    vaDof *dof);

//---- vaSkelprot ----------------------------------------------------------//
extern vaSkelprot  *vaSkelprotNew( char *name);
extern void    vaSkelprotDelete(          vaSkelprot *skp);
extern void   vaSkelprotSetOffsetPos(  vaSkelprot *skp, float x, float y, float z);
extern void   vaSkelprotSetOffsetHpr(  vaSkelprot *skp, float h, float p, float r);
extern void   vaSkelprotAddChild(       vaSkelprot *skp, vaSkelprot *child);
extern void   vaSkelprotSetRzDOF(      vaSkelprot *skp, vaDof *dof);
extern void   vaSkelprotSetRyDOF(      vaSkelprot *skp, vaDof *dof);
extern void   vaSkelprotSetRxDOF(      vaSkelprot *skp, vaDof *dof);
extern void   vaSkelprotSetTxDOF(      vaSkelprot *skp, vaDof *dof);
extern void   vaSkelprotSetOffDist(    vaSkelprot *skp, float distance);
extern int     vaSkelprotGetIndex(       vaSkelprot *skp);

//---- vaActclass ----------------------------------------------------------//
vaActclass *          vaActclassNew(char *name);
void                    vaActclassDelete(vaActclass *actclass);
void                    vaActclassAddRootSkelprot(vaActclass *actclass, vaSkelprot *skp);

void                     vaActclassCompile(          vaActclass *actclass);
void                     vaActclassPrint(              vaActclass *actclass);
```

```
extern int              vaActclassGetNumActors( vaActclass *ac);
extern vaActor *    vaActclassGetActor(      vaActclass *ac, int actorIndex);

extern void   vaActclassSetBehaviourData(   vaActclass *ac,void *data);
extern void * vaActclassGetBehaviourData(  vaActclass *ac);

extern void   vaActclassSetActorBehaDataCreateFunc(vaActclass *ac, vaActorBehaDataCreateFunc func);
extern void   vaActclassSetActorBehaDataDeleteFunc(vaActclass *ac, vaActorBehaDataDeleteFunc func);

extern void   vaActclassSetRefpointBehaviourFunc(   vaActclass *ac,    vaActorBehaFunc func);
extern void   vaActclassSetSklsrootBehaviourFunc(   vaActclass *ac,    vaActorBehaFunc func);
extern void   vaActclassSetDofsBehaviourFunc(       vaActclass *ac,    vaActorBehaFunc func);

extern void   vaActclassSetDrawFuncs(           vaActclass *ac,     vaActorDrawFunc preDrawFunc,
                                                                    vaActorDrawFunc postDrawFunc);

extern void   vaActclassSetSklsrootBsphere(     vaActclass *ac,     float4 sph);

extern int    vaActclassFindSklIndexByName(     vaActclass *ac,     char *sklName);
extern int    vaActclassFindDofIndexByName(     vaActclass *ac,     char *dofName);

extern void   vaActclassGetDofBounds(           vaActclass *ac,     int dofIndex, float *min, float *max);
extern void   vaActclassGetDofRanges(           vaActclass *ac,     int dofIndex, float *rin, float *rmax, float *def);

extern float  vaActclassGetSkpOffDist(          vaActclass *ac,     int skpIndex);

extern int      vaActclassAddExtension(         vaActclass *ac,     vaExtension *ext,  int slotIndex);
extern void * vaActclassGetExtData(             vaActclass *ac,     int slotIndex);
extern void * vaActclassGetDofExtData(          vaActclass *ac,     int dofIndex, int slotIndex);
extern void * vaActclassGetSkpExtData(          vaActclass *ac,     int skpIndex, int slotIndex);
```

En el anterior bloque de código se describen las principales funciones de creación de una "Clase de Actor". Como se puede observar está dividido en tres zonas, la primera de ellas dedicada a la gestión de las estructuras *vaDof*, la segunda a las estructuras *vaSkelprot,* y la última, al manejo de la estructura *vaActclass*.

Con respecto a las funciones del *vaDof,* las funciones ***vaDofNew()*** y ***vaDofDelete()*** sirven para crear y borrar respectivamente una estructura de este tipo**.** Las funciones ***vaDofSetMinMax()*** y ***vaDofSetRanges()***, sirven para fijar los topes mínimo y máximo de ese grado de libertad, y también, el rango de trabajo cómodo, y el valor de descanso. Mediante la función ***vaDofGetIndex()*** se obtiene el índice asignado a ese *vaDof* dentro de la clase de actor, previo a la obtención del índice es necesario haber llamado a la función *vaActclassCompile* que será descrita posteriormente.

En relación con la estructura *vaSkelprot*, las funciones ***vaSkelprotNew()*** y ***vaSkelprotDelete()*** sirven respectivamente para su creación y borrado. Las funciones ***vaSkelprotSetOffsetPos()*** y ***vaSkelprotSetOffsetHpr()*** sirven para fijar sus offsets de traslación y orientación. Las funciones ***vaSkelprotSetRzDOF()***, ***vaSkelprotSetRyDOF()***, ***vaSkelprotSetRxDOF()*** ***vaSkelprotSetTxDOF()***, sirven para fijar los grados de libertad que tendrá ese punto de articulación. La función ***vaSkelprotAddChild()*** se emplea para definir una estructura jerárquica, conectando unos puntos de articulación con otros. La función ***vaSkelprotSetOffDist()*** permite asignar la distancia a la que un punto de articulación deja de ser

procesado, esta función es empleada, por tanto, en la definición de los niveles de detalle topológicos. Por último, la función ***vaSkelprotGetIndex()*** retorna el índice de ese *vaSkelprot* dentro de la tabla de *vaSkelprot* de la *vaActclass*, para poder llamar a esta función, es necesario haber llamado con anterioridad a la función *vaActclassCompile().*

Con respecto a las funciones que afectan a la estructura *vaActclass*, ***vaActclassNew()*** y ***vaActclassDelete()*** permiten crear una nueva *vaActclass* y borrarla respectivamente. La función ***vaActclassAddRootSkelprot()*** permite añadir los *vaSkelprots* que actúan como padres de la jerarquía que define la estructura articulada (esta función se emplea junto con *vaSkelprotAddChild()* para definir la estructura esquelética de los actores de esta clase). La función ***vaActclassCompile()***, citada anteriormente, genera las tablas de índices a *vaDof* y *vaSkelprot*, también comprueba si ha habido incoherencias a la hora de establecer relaciones entre los *vaSkelprots,* o entre ellos y sus *vaDofs*. La función ***vaActclassPrint()*** imprime la organización interna del *vaActclass* de forma que se pueda comprobar si todo ha sido definido de la forma deseada.

Como ya se ha detallado anteriormente, un *vaActclass* tiene una tabla de punteros a sus actores, mediante las funciones ***vaActclassGetNumActors()*** y ***vaActclassGetActor()*** se puede conocer el número total de actores de esa clase de actor, y acceder a alguno de ellos a través de su índice.

La funciones ***vaActclassSetBehaviourData()*** y ***vaActclassSetActorBehaDataCreateFunc(),*** permiten fijar los datos de usuario que se quiere almacenar dentro de la estructura *vaActclass* y dentro de la estructura *vaActor* respectivamente. Como se puede observar, los datos de usuario almacenados dentro del *vaActor* se fijan de un modo indirecto mediante el uso de una función de creación. Esto facilita que un actor pueda ser creado o borrado de una forma automática. La función ***vaActclassGetBehaviourData()*** retorna los datos de usuario de una *vaActclass,* y la función ***vaActclassSetActorBehaDataDeleteFunc()*** permite fijar la función de borrado de los datos de usuario de un *vaActor.*

Las funciones ***vaActclassSetRefpointBehaviourFunc()***, ***vaActclassSetSklsrootBehaviourFunc()*** y ***vaActclassSetDofsBehaviourFunc()*** sirven para fijar las funciones que se ejecutaran para actualizar los valores del *Reference Point*, el *Skeletons Root,* o los valores de los grados de libertad de un actor virtual. Las funciones de gestión del comportamiento de un actor están divididas en tres bloques para que los métodos de gestión de *cull* de los actores virtuales llamen solamente a las funciones que sean necesarias. La función ***vaActclassSetDrawFuncs()*** permite indicar unas funciones de callback que serán llamadas antes y después de dibujar un actor virtual, mediante estas funciones se pueden cambiar algunas propiedades de dibujado del actor (por ejemplo su textura o su color).

La función *vaActclassSetSklsrootBsphere()* permite fijar *bounding sphere* definida en el sistema de coordenadas del *Skeletons Root*. Esta *bounding sphere* suele ser común a todos actores de la misma clase, pero puede ser redefinida en cada actor en particular si es necesario.

Las funciones ***vaActclassFindSklIndexByName()*** y ***vaActclassFindDofIndexByName()***, sirven para obtener el índice de un *vaSkelprot* o un *vaDof* a través de su nombre.

Las funciones ***vaActclassGetDofBounds()*** y ***vaActclassGetDofRanges()*** permiten conocer los valores mínimo y máximo de un grado de libertad, y también el rango de trabajo no forzado y el valor de descanso. La función ***vaActclassGetSkpOffDist()*** retorna la distancia de desactivación de un punto de articulación. Se puede observar como las funciones anteriores emplean los índices para acceder a las estructuras *vaDof* o *vaSkelprot* deseadas.

La función ***vaActclassAddExtension()*** permite añadir una extensión a un *vaActclass,* indicando, además, el índice del slot en el que se quiere que esté almacenada. La función ***vaActclassGetExtData()*** retorna un puntero a los datos de una extensión almacenados en el *vaActclass*. Las funciones ***vaActclassGetDofExtData()*** y ***vaActclassGetSkpExtData()*** retornan un puntero a los datos de un *extension slot* almacenados en un *vaDof* o en un *vaSkelprot* de una *Clase de Actor.*

```
//---- vaExtension ------------------------------------------------------//
typedef void   (*vaExtRuntimeFunc)(  vaActor *act);
typedef void * (*vaExtDataCreateFunc)(void *data);
typedef void   (*vaExtDataDeleteFunc)(void *data);

extern vaExtension * vaExtensionNew( char *name);
extern void          vaExtensionDelete(     vaExtension *ext);

extern void vaExtensionSetCreateFuncs(        vaExtension *ext,   vaExtDataCreateFunc actclassFunc,
                                                                  vaExtDataCreateFunc actorFunc,
                                                                  vaExtDataCreateFunc dofFunc,
                                                                  vaExtDataCreateFunc skpFunc);

extern void vaExtensionSetDeleteFuncs(        vaExtension *ext,   vaExtDataDeleteFunc actclassFunc,
                                                                  vaExtDataDeleteFunc actorFunc,
                                                                  vaExtDataDeleteFunc dofFunc,
                                                                  vaExtDataDeleteFunc skpFunc);

extern void vaExtensionSetActorUpdateFunc(    vaExtension *ext,   vaExtRuntimeFunc func);
```

Las funciones ***vaExtensionNew()*** y ***vaExtensionDelete()*** permiten crear una nueva extensión, y también eliminarla. La nueva extensión es incorporada dentro de la lista de extensiones gestionadas por la librería mediante la función ***vaExtensionAddToList()***, esta función retorna el índice de la extensión dentro de esa tabla, este valor actúa como identificador único.

Una extensión necesita almacenar sus propios datos dentro de los *extension slots* de las estructuras *vaActclass*, *vaActor*, *vaDof* y *vaSkp*. El contenido de los *extension slots* es definido mediante la asignación a la extensión de unas funciones para la creación de estos datos, estas funciones son fijadas mediante ***vaExtensionSetCreateFuncs()***. De modo similar, las funciones de liberación de la memoria ocupada en los *extension slots,* son definidas mediante la función ***vaExtensionSetDeleteFuncs()***.

Cuando una extensión a sido asignada a una clase de actor, la librería de actores se encarga, en cada frame y para cada uno los actores afectados, de llamar a una función de actualización de la información relativa a esa extensión. Esta función de actualización es fijada mediante ***vaExtensionSetActorUpdateFunc()***.

```
//==== ACTOR ================================================================//
//---- vaActor -----------------------------------------------------------//
extern vaActor *     vaActorNew(    vaActclass *ac, char *name);
extern vaActor *     vaActorClone(  vaActor *act, char *name);
extern int           vaActorDelete( vaActor *act);

extern void     vaActorSetRefpointPos(     vaActor *act, float3 pos);
extern void     vaActorGetRefpointPos(     vaActor *act, float3 pos);
extern void     vaActorSetRefpointHpr(     vaActor *act, float3 hpr);
extern void     vaActorGetRefpointHpr(     vaActor *act, float3 hpr);

#define VADOF_SKLSROOT_PX       -6
#define VADOF_SKLSROOT_PY       -5
#define VADOF_SKLSROOT_PZ       -4
#define VADOF_SKLSROOT_H        -3
#define VADOF_SKLSROOT_P        -2
#define VADOF_SKLSROOT_R        -1

extern void     vaActorSetDofValue(                vaActor *act, int dofIndex,  float value);
extern void     vaActorSetDofValueSpeed(           vaActor *act, int dofIndex,  float endValue, float speed,
                                                   float deltaTime);
extern void     vaActorSetDofValueGradual(         vaActor *act, int dofIndex, float endValue,float normValue);
extern float    vaActorGetDofValue(                vaActor *act, int dofIndex);

extern float       vaActorGetEyeDistance(          vaActor *act);
extern vaActclass * vaActorGetActclass(            vaActor *act);
extern int         vaActorGetIndex(                vaActor* act);

extern void     vaActorSetBehaviourData(           vaActor *act, void *data);
extern void *   vaActorGetBehaviourData(           vaActor *act);
extern void     vaActorSetRefpointBehaviourFunc( vaActor *act,       vaActorBehaFunc func);
extern void     vaActorSetSklsrootBehaviourFunc( vaActor *act,       vaActorBehaFunc func);
extern void     vaActorSetDofsBehaviourFunc(     vaActor *act,       vaActorBehaFunc func);
extern void     vaActorSetDrawFuncs(               vaActor *act,       vaActorDrawFunc preDrawFunc,
                                                                       vaActorDrawFunc postDrawFunc);

extern void     vaActorSetSklsrootBsphere( vaActor *act, float4 sph);
extern void     vaActorSetRefpointBsphere( vaActor *act, float4 sph);

extern void  vaActorSetSklTracking(      vaActor *act, int sklIndex, int mode);
extern int   vaActorGetSklTracking(      vaActor *act, int sklIndex);
extern void  vaActorGetSklAbsMatrix(     vaActor *act, int sklIndex, float16 mat);
extern void  vaActorGetSklAbsPosHpr(  vaActor *act, int sklIndex,  float3 absPos, float3 absHpr);
extern void  vaActorGetSklOffsetMat(     vaActor *act, int sklIndex, float16 offsetMat);

extern void * vaActorGetExtData(     vaActor *act, int slotIndex);

//---- Funciones para ensamblar el nodo Actor---------------------------------//
extern void *       vaActorGetSgNode(   vaActor *act);
void        vaActorAddChildSgNode(     vaActor *act, void *sgNode);
void        vaActorSklAddChildSgNode( vaActor *act, int sklIndex,void *sgNode);
```

La función ***vaActorNew()*** crea un nuevo actor de una clase determinada y le asigna un nombre. La función ***vaActorClone()*** sirve para crear un nuevo actor que es una copia de uno ya existente, permitiendo

la asignación de un nombre diferente. La función ***vaActorDelete()*** libera la memoria ocupada por un actor.

Las funciones ***vaActorSetRefpointPos()***, ***vaActorSetRefpointHpr()***, ***vaActorGetRefpointPos()*** y ***vaActorGetRefpointHpr()***, permiten fijar y consultar los valores de posición y orientación del *Reference Point* de un determinado actor.

Las funciones ***vaActorSetDofValue()***, ***vaActorSetDofValueSpeed(), vaActorSetDofValueGradual()*** y ***vaActorGetDofValue()***, permiten fijar y consultar el valor de un determinado grado de libertad. Los grados de libertad son identificados mediante índices. La posición y orientación del *Skeletons Root* es gestionada también desde estas funciones, empleando para ello los índices especiales ***VADOF_SKLSROOT_PX***, ***VADOF_SKLSROOT_PY***, ***VADOF_SKLSROOT_PZ***, ***VADOF_SKLSROOT_H***, ***VADOF_SKLSROOT_P*** y ***VADOF_SKLSROOT_R***. La función *vaActorSetDofValueSpeed()* y *vaActorSetDofValueGradual(),* permiten fijar el valor de un grado de libertad de una forma progresiva, la primera pasando como parámetros el valor final que alcanzará, la velocidad a la que se mueve hacia ese valor, y el tiempo transcurrido desde que comenzó la transición, y la segunda indicando directamente el valor final, y un valor entre 0.0 y 1.0 que indica el punto de la transición.

La función ***vaActorGetEyeDistance()*** se emplea para conocer la distancia del actor virtual a la cámara. La función ***vaActorGetActclass()*** retorna un puntero a la clase de actor a la que pertenece un actor, y la función ***vaActorGetIndex()*** retorna el índice de un actor dentro de la lista de actores de su *vaActclass*.

Las funciones ***vaActorSetBehaviourData()*** y ***vaActorGetBehaviourData()*** acceden a los datos de usuario que personalizan el comportamiento de un actor. De igual modo las funciones ***vaActorSetRefpointBehaviourFunc()***, ***vaActorSetSklsrootBehaviourFunc()*** y ***vaActorSetDofsBehaviourFunc()***, definen las funciones que han de ser llamadas para gestionar su comportamiento. La función ***vaActorSetDrawFuncs()*** permite fijar unas rutinas que serán llamadas justo antes y después del dibujado de un actor virtual. Tanto los datos de usuario como las funciones de callback del comportamiento pueden ser fijadas desde la clase de actor mediante las funciones *vaActclassSetActorBehaDataCreateFunc()*, *vaActclassSetRefpointBehaviourFunc()*, *vaActclassSetSklsrootBehaviourFunc(),* *vaActclassSetDofsBehaviourFunc()*, y *vaActclassSetDrawFuncs()* descritas anteriormente, pero las del actor tienen preferencia y permite que actores de la misma clase presenten comportamientos muy diferentes.

Las funciones ***vaActorSetSklsrootBsphere()*** y ***vaActorSetRefpointBsphere()*** permiten fijar las *bounding spheres* empleadas en el *culling* de actores. La *SklsRoot Bsphere* también puede ser fijada desde la función *vaActclassSetSklsrootBsphere()*, pero tiene prioridad la fijada desde el propio actor.

La función ***vaActorSetSklTracking()*** permite fijar el flag de tracking de un determinado nodo *Skeleton*, ello permite conocer sus valores de orientación y posición en coordenadas del mundo. La función

vaActorGetSklTracking() permite consultar si el tracking está o no activo. La función ***vaActorGetSklAbsMatrix()*** retorna la matriz absoluta correspondiente con ese nodo *Skeleton*; la función ***vaActorGetSklAbsPosHpr()***, la posición y orientación indicadas con respecto al sistema de coordenadas del mundo, y la función ***vaActorGetSklOffsetMat()***, la matriz absoluta sin tener en cuenta las transformaciones aplicadas por los grados de libertad de ese nodo *Skeleton*.

La función ***vaActorGetExtData()*** retorna un puntero a los datos de una extensión almacenados en el *vaActor.* Por último, las funciones ***vaActorGetSgNode()***, ***vaActorAddChildSgNode()*** y ***vaActorSklAddChildSgNode()***, permiten conectar nodos normales del grafo de escena a la jerarquía del actor, y conectar al actor en cualquier punto de la jerarquía global de la escena. La primera función retorna un puntero al nodo del grafo de escena que se corresponde con el nodo *Actor*, la segunda permite colgar un nodo genérico del nodo *Actor* y la tercera permite colgar un nodo genérico de uno de sus nodos *Skeleton*.

```
//---- vaLod ------------------------------------------------------------------//
extern vaLod *      vaLodNew(char *name, vaActor *act);
extern vaLod *      vaLodCopy(          vaLod *orig);
extern void         vaLodAddChild(     vaLod *lod, void *sgNode, float distance);
extern int          vaLodSelectChild(  vaLod *lod, float distance);
extern void *       vaLodGetSgNode(  vaLod *lod);
```

La función ***vaLodNew()*** permite crear una nueva estructura de tipo *vaLod*, siendo necesario indicarle el actor al que va vinculada, y el nombre que se desea que tenga. Las funciones ***vaLodDelete()*** y ***vaLodCopy()*** sirven para borrar o realizar una copia de un *vaLod*. La función ***vaLodAddChild()*** añade como hijo un nodo de tipo genérico, definiendo, además, su distancia de desactivación. La función ***vaLodSelectChild()*** permite seleccionar directamente una de las ramas hijas mediante la distancia, y por último la función ***vaLodGetSgNode()***, retorna un puntero al nodo del grafo de escena que actúa como contenedor para este *vaLod*.

```
//--- Scene graph ------------------------------------------------------------//
extern void * vaSgMalloc(size_t nbytes);
extern void * vaSgRealloc(void *ptr, size_t nbytes);
extern void   vaSgFree(void *ptr);

extern void * vaSgGrpNodeCreate(char *name);
extern void   vaSgGrpNodeAddChild(              void *group, void *child);
extern void   vaSgGrpNodeInsertChild(           void *group, int index, void *child);
extern void   vaSgGrpNodeRemoveChild(           void *group, void *child);
extern int    vaSgGrpNodeGetNumChildren(        void *group);
extern void * vaSgGrpNodeGetChild(              void *group, int nChild);
```

Las funciones ***vaSgMalloc()***, ***vaSgRealloc()*** y ***vaSgFree()***, permiten hacer que la memoria usada para las estructuras de datos empleadas por la librería de actores virtuales, sea gestionada con las mismas funciones que emplea el grafo de escena.

Las funciones ***vaSgGrpNodeCreate()***, ***vaSgGrpNodeAddChild()***, ***vaSgGrpNodeRemoveChild()***, ***vaSgGrpNodeGetNumChildren()*** y ***vaSgGrpNodeGetChild()***, definen funciones genéricas de un grafo de escena, permiten organizar los nodos que componen un actor virtual, y también ubicar al actor virtual en cualquier punto del grafo de escena general, sin que para ello sea necesario utilizar las funciones proporcionadas por el grafo de escena concreto. Esto garantiza la portabilidad de una aplicación que use esta librería entre diferentes grafos de escena.

```
//---- Estadisticas ------------------------------------------------------//
#define VASTATS_DISABLE_NONE                0x00
#define VASTATS_DISABLE_REFPOINTBEHAVIOUR   0x01
#define VASTATS_DISABLE_SKLSROOTBEHAVIOUR   0x02
#define VASTATS_DISABLE_DOFSBEHAVIOUR       0x04
#define VASTATS_DISABLE_FULLBEHAVIOUR       0x07
#define VASTATS_DISABLE_ACTORTRAVERSE       0x10
#define VASTATS_DISABLE_MATRIXAPPLY         0x20
#define VASTATS_DISABLE_DRAWING             0x40

extern void     vaStatsSetTestCullProcess(int mode);
extern int      vaStatsGetTestCullProcess();
extern void     vaStatsSetViewBSpheres(int mode);
extern int      vaStatsGetViewBSpheres();

extern void     vaStatsSetMode(int mode, int value);
extern int      vaStatsGetMode(int mode);
extern void     vaStatsShow();
#endif
```

Para finalizar, en este último del fichero "*vaApi.h"*, se muestran los prototipos de distintas funciones que permiten depurar y ajustar el rendimiento de la aplicación. La función ***vaStatsSetTestCullProcess()***, permite evitar la comprobación de culling, de modo que se dibujen todos los actores, estén o no dentro del frustum. La función ***vaStatsSetViewBSpheres()*** representa la *SkelRoot Bsphere* y *RefPoint Bsphere* de cada actor virtual mediante un par de esferas dibujadas en modo alámbrico. La función ***vaStatsSetMode()*** permite desactivar alguno los procesos relacionados con la gestión de los actores, el significado de los modos que se puede pasar a esta función es el siguiente: *VASTATS_DISABLE_REFPOINTBEHAVIOUR* desactiva la gestión de comportamiento relacionada con el cálculo de la posición del *Reference Point* de los actores; *VASTATS_DISABLE_SKLSROOTBEHAVIOUR* desactiva los cálculos relacionados con la gestión de comportamiento de los *Skeleton Roots* de los actores; *VASTATS_DISABLE_DOFSBEHAVIOUR*, desactiva la gestión de comportamiento relacionada con la actualización de los valores de los grados de libertad; *VASTATS_DISABLE_FULLBEHAVIOUR* desactiva toda la gestión de comportamiento; *VASTATS_DISABLE_ACTORTRAVERSE* desactiva el recorrido de los nodos que componen internamente al actor virtual; *VASTATS_DISABLE_MATRIXAPPLY* desactiva la aplicación de matrices de transformación; *VASTATS_DISABLE_DRAWING* desactiva el dibujado de la geometría de los actores virtuales y por último, mediante *VASTATS_DISABLE_NONE* se restauran los aspectos desactivados.

Por último ***vaStatsShow()*** imprime en una shell información referente a la forma en la que se está realizando el procesado de los actores.

8.5 Método de integración en un grafo de escena genérico.

Uno de los principales objetivos de esta Tesis Doctoral es la búsqueda de un método estándar que simplifique el proceso de integración de actores virtuales en una aplicación de simulación. Por esta razón ha existido la preocupación de describir un API que actúe empleando como base la filosofía de trabajo de un grafo de escena tradicional, de igual modo, también ha existido la intención de hacer un desarrollo que no estuviese vinculado a ningún grafo de escena en particular. Si bien ejemplo de aplicación final se ha integrado sobre el grafo de escena *OpenGl Performer*, cualquier desarrollo con actores virtuales basado en las estructuras y métodos propuestos en este trabajo, sería fácilmente integrado en cualquier otro grafo de escena.

En los siguientes subapartados se describe la forma en la que se realiza la integración entre el grafo de escena particular y la librería de gestión de actores virtuales, así como las características mínimas de un grafo de escena genérico para que dicha integración sea posible. Por último, se dedica un apartado especial a mostrar la forma en la que se realiza la integración de la gestión del comportamiento y culling de los actores virtuales, dentro las funciones de callback de los nodos del grafo de escena.

8.5.1 API de configuración de la librería de actores para su integración en un grafo de escena genérico (fichero "vaSgCfg.h").

Una de las ideas principales de la utilización de esta librería de actores, es que un programador pueda definir los actores virtuales y su comportamiento de un modo independiente al grafo de escena (utilizando para ello las funciones del API mostradas en el fichero ***"vaApi.h"*), y** finalmente pueda adaptar dicho desarrollo para su funcionamiento sobre un grafo de escena en particular, empleando para ello una serie de funciones de configuración. Estas funciones que configuran a la librería de actores para trabajar con ese grafo de escena se encuentran recopiladas en el fichero "***vaSgCfg.h"***, y su contenido es el siguiente:

```
#ifndef __VA_SGCFG_H__
#define __VA_SGCFG_H__

//--- Tipos de funciones empleados en la comunicacion con el grafo de escena. ----------------------------------//
//-------------------------------------------------------------------------------------------------------------------//
//- -Funciones de gestion de la memoria. ------------------------------------------------//
typedef void  * (*vaSgMallocFunc)(size_t nbytes);
typedef void  * (*vaSgReallocFunc)(void *ptr, size_t nbytes);
typedef void    (*vaSgFreeFunc)(void *ptr);

//--- Funciones de control de nodos de tipo grupo. ----------------------------------------//
typedef void *  (*vaSgGrpNodeCreateFunc)(char *name);
typedef void    (*vaSgGrpNodeDeleteFunc)(void *grpNode);
typedef void    (*vaSgGrpNodeAddChildFunc)(void *parent, void *child);
typedef void    (*vaSgGrpNodeInsertChildFunc)(void *parent, int index, void *child);
typedef void    (*vaSgGrpNodeRemoveChildFunc)(void *parent, void *child);
typedef int     (*vaSgGrpNodeGetNumChildrenFunc)(void *grpNode);
typedef void *  (*vaSgGrpNodeGetChildFunc)(void *grpNode, int nChild);
//-- Funciones de control de nodos de tipo Switch ----------------------------------------//
typedef void * (*vaSgSwNodeCreateFunc)(char *name);
```

```
typedef void    (*vaSgSwNodeDeleteFunc)(void *swNode);
typedef void   (*vaSgSwNodeSelectChildFunc)(void *parent, float child);

//-- Funciones de gestion de los datos de usuario de un nodo ----------------------------//
typedef void   (*vaSgNodeSetUDataFunc)(void *sgNode, void *udata);
typedef void * (*vaSgNodeGetUDataFunc)(void *sgNode);

//--Función de fjiación de rutinas de callback en un nodo ----------------------------------//
typedef void   (*vaSgNodeSetCbFuncsFunc)(void *sgNode);

//-- Función de acceso al tiempo global de las simulación ----------------------------------//
typedef float  (*vaSgGetTimeFunc)(void);

//-- Función que asigna  una bounding sphere sobre un nodo  ----------------------------//
typedef void   (*vaSgApplyBSphToNodeFunc)(void *sgNode, float4 sph);

//-- Función que retorna el efecto del culling del grafo de escena sobre un nodo-------//
typedef int    (*vaSgGetCullResultFunc)(void *sgNode, void *data);

//-- Función que permite realizar clonar una rama del grafo de escena  ------------------//
typedef void * (*vaSgCloneTreeFunc)(void *sgNode);

//--- Funciones de configuraración de la librería Va para un  grafo de escena genérico. -------------------------//
//-----------------------------------------------------------------------------------------------------------------------------------//
void   vaSgSetMemoryFuncs(          vaSgMallocFunc    mallocFunc,  vaSgReallocFunc reallocFunc,
                                    vaSgFreeFunc      freeFunc);

void   vaSgSetGetTimeFunc(          vaSgGetTimeFunc                      getTimeFunc);

void   vaSgSetGroupNodeFuncs(       vaSgGrpNodeCreateFunc                groupCreate,
                                    vaSgGrpNodeDeleteFunc                groupDelete,
                                    vaSgGrpNodeAddChildFunc              groupAddChild,
                                    vaSgGrpNodeInsertChildFunc           groupInsertChild,
                                    vaSgGrpNodeRemoveChildFunc           groupRemoveChild,
                                    vaSgGrpNodeGetNumChildrenFunc        groupGetNumChildren,
                                    vaSgGrpNodeGetChildFunc              groupGetChild);

void   vaSgSetSwitchNodeFuncs(      vaSgSwNodeCreateFunc                 switchCreate,
                                    vaSgSwNodeDeleteFunc                 switchDelete,
                                    vaSgSwNodeSelectChildFunc            switchSelectChild);

void   vaSgSetNodeUserDataFuncs(    vaSgNodeSetUDataFunc                 setUData,
                                    vaSgNodeGetUDataFunc                 getUData);

void   vaSgSetApplyBsphToNodeFunc(vaSgApplyBSphToNodeFunc                bsphToNode);

void   vaSgSetCloneTreeFunc(        vaSgCloneTree                        cloneTree);

void   vaSgSetNodeCullResultFunc( vaSgGetCullResultFunc                  getCullResult);

void   vaSgSetNodesCbFuncs(         vaSgNodeSetCbFuncsFunc               setActNodeCbFuncs,
                                    vaSgNodeSetCbFuncsFunc               setSklNodeCbFuncs,
                                    vaSgNodeSetCbFuncsFunc               setLodNodeCbFuncs);

//-------funciones asociadas a los callback de los nodos, ------------------------------------------//
#define VA_PRECULLRESULT_CONT                   0
#define VA_PRECULLRESULT_CONT_ALLIN             1
#define VA_PRECULLRESULT_PRUNE                  2
int    vaActNodePreCull(    void *sgNode, float16 sgInitialMat, int refpointBsphIn, void *data);
void   vaActNodePreDraw(    void *actNode);
void   vaActNodePostDraw(   void *actNode);
```

```
int     vaSklNodePreCull(     void *sklNode,  void *data);
void    vaSklNodePreDraw(     void *sklNode);
void    vaSklNodePostDraw(    void *sklNode);

void    vaLodNodePrecull(     void *lodNode, void *data);
#endif
```

La función ***vaSgSetMemoryFuncs()*** permite fijar las rutinas de callback que serán empleadas para hacer un malloc, un free o un realloc, usando el mismo tipo de gestión de memoria que es empleado por grafo de escena.

La función ***vaSgSetGetTimeFunc()*** permite fijar la rutina que será llamada para conocer el tiempo global de la simulación.

La función ***vaSgSetGroupNodeFuncs()*** permite fijar las rutinas de callback que se emplearán para crear un nodo de tipo "Grupo", también para borrarlo y para gestionar sus hijos. La función ***vaSgSetSwitchNodeFuncs()*** define las rutinas de callback a emplear para crear y borrar un nodo de tipo "Switch", y también para indicar cual es su rama activa. La función ***vaSgSetNodeUserDataFuncs()*** permite fijar y obtener los datos de usuario de un nodo.

La función ***vaSgSetApplyBsphToNodeFunc()*** define la rutina de callback mediante la cual es posible fijar una determinada *bounding sphere* sobre un nodo. Esa *bounding sphere* será empleada durante el proceso de culling del grafo de escena.

La función ***vaSgSetCloneTreeFunc()*** define una rutina de callback mediante la cual se realizar una copia de un trozo de grafo de escena.

La función ***vaSgSetNodeCullResultFunc()*** define una rutina de callback mediante la cual se puede conocer la forma en la que el proceso de culling tradicional está afectando a un determinado nodo.

Por último, la rutina ***vaSgSetNodesCbFuncs()*** permite definir cuales serán las distintas rutinas de callback que serán llamadas durante el procesado del grafo de escena. Ésta es la parte de la integración con el grafo de escena que resulta más compleja, puesto que en ella se ven implicados los métodos especiales de gestión de cull y lod específicos para actores virtuales. Las funciones de callback asociadas a los nodos de tipo *Actor* han de hacer internamente llamadas a las funciones ***vaActNodePreCull()***, ***vaActNodePreDraw()*** y ***vaActNodePostDraw()***, las asociadas a un nodo *vaSkeleton* han de usar internamente las funciones ***vaSklNodePreCull()***, ***vaSklNodePreDraw()*** y ***vaSklNodePostDraw()***. De igual modo la asociada a un nodo *vaLod* ha de llamar a la función ***vaLodNodePrecull()***.

La forma en la que han de estar implementadas estas funciones de callback es explicada más detalladamente en el apartado 8.5.3, de igual modo, también puede ser observada su implementación práctica en el caso de *OpenGl Performer* que será presentada en el siguiente capítulo.

8.5.2 Requisitos mínimos del grafo un de escena para poder ser empleado con esta librería.

Los requisitos mínimos que un grafo de escena genérico ha de cumplir para poder ser integrado con los desarrollos de esta tesis son los siguientes:

- Existencia de **nodos de tipo Grupo** y **nodos de tipo Switch**. Es decir, de un nodo que pueda tener un número determinado de hijos, y de uno que también pueda tener varios hijos pero que disponga de un de un parámetro que permita seleccionar cual es el hijo activo.
- Capacidad para definir **datos de usuario** en un nodo del grafo de escena.
- Posibilidad de personalización en la forma en la que el grafo de escena realiza las operaciones de culling. Para ello es necesario poder para asignar a los nodos **rutinas de callback** que se ejecuten automáticamente durante el proceso de recorrido del grafo de escena -*OpenGl Performer* permite hacer esto mediante la rutina *pfNodeTravFuncs()*-. Estas rutinas han de poder actuar sobre el proceso de cull, forzando a que se comporte de un modo diferente al estándar (en *OpenGL Performer* el valor retornado por la rutina de pre-cull permite indicar si el subgrafo que depende de ese nodo ha de ser o no dibujado). También capacidad para consultar sobre el efecto del culling propio del grafo de escena sobre un nodo -labor llevada a cabo por la función *pfGetCullResult()* en *Opengl Performer*-, y capacidad para poder desactivar la comprobación de culling en los nodos de un subgrafo de escena -misión llevada a cabo en *OpenGl Performer* mediante el comando *pfCullResult(PFIS_ALL_IN)* -.
- Capacidad para fijar rutinas de callback que se ejecuten en los instantes anteriores y posteriores al procesado de un nodo para su dibujado -operación realizada en *OpenGl Performer* también mediante la orden *pfNodeTravFuncs()*-.
- Capacidad para consultar la matriz absoluta de un nodo durante el procesado el grafo de escena -esto es logrado en *OpenGl Performer* mediante el comando *pfGetTravMat()*-.
- Capacidad para fijar la ***bounding sphere*** de un nodo de forma externa -en *OpenGl Performer* se emplea la orden *pfNodeBSphere()*-.
- Capacidad para copiar de una forma rápida un subgrafo de escena, -operación realizada en OpenGl Performer mediante el comando *pfClone()*-.

8.5.3 Gestión de las funciones de callback a los nodos.

Los nodos *vaActor*, *vaSkeleton* y *vaLod* tienen rutinas de callback de pre-cull, es decir, que se llaman durante el recorrido del grafo de escena y permiten personalizar la forma en la que se realiza el culling en estos nodos.

En el *nodo vaActor*, el objetivo del la llamada *pre-cull* es realizar una llamada a la rutina *vaActNodePreCull(),* la cual se encarga de calcular la relación entre la *refpointBsph* y el *frustum*, y también la distancia entre el actor y la cámara, la cual será empleada para la gestión de nivel de detalle. También es misión de esta función llamar a las funciones de gestión de comportamiento, y actuar sobre los valores del *Skeletons Root,* hacer la comprobación de culling con la *sklsroot Bsphere* y actualizar los

grados de libertad del actor en el caso de que sea necesario. A esta función ha de pasársele como parámetro un puntero al nodo, el valor de la matriz inicial que tiene ese nodo por estar colocado en un punto determinado del grafo de escena (*sgInitialMatrix*), y también un puntero a alguna estructura que le pueda proporcionar información sobre el gestor de culling del grafo de escena. Esta función retorna un entero que informa como ha de seguir realizándose el culling del subgrafo que depende de ese nodo, los posibles valores retornados son:

- **VA_PRECULLRESULT_PRUNE**: El nodo y el subgrafo de escena que depende de él no han de ser procesados. Están totalmente fuera de la pirámide de visión.
- **VA_PRECULLRESULT_CONT**: Este nodo y su subgrafo han de ser procesados. Se realizarán comprobaciones de culling con las *Internal Bspheres*.
- **VA_PRECULLRESULT_CONT_ALLIN:** Este nodo y su subgrafo han de ser procesados, pero se ha de deshabitar la comprobación de culling puesto que se encuentra totalmente dentro de la pirámide de visión.

La forma en la que las rutinas de callback y el gestor del culling del grafo de escena interactúan puede ser muy diferente de unos grafos de escena a otros. El siguiente bloque de código intenta representar de una forma genérica la organización interna de la rutina de *precull* del nodo Actor. Con la intención de hacer más sencilla la explicación se está haciendo la suposición de que el grafo de escena proporciona una estructura de nombre *cullManager,* mediante la cual podemos interactuar con su gestión de culling, los nombres de las funciones relacionadas con el *cullManager* son autoexplicativas:

```
//Obtener el puntero a la estructura que proporciona información sobre el culling.
void *cullMng = sgGetCullManager()

//Obtener el puntero al nodo que esta llamando a esta función de callback.
void *sgNode = cullManagerGetCurNode(cullMng);

//obtener la matriz actual en este momento del recorrido del grafo de escena.
float16 sgInitialMat = cullManagerGetCurMatrix(cullMng);

//LLamar a la rutina principal de gestion del nodo Actor.
int precullResult = vaActNodePreCull( sgNode, sgInitialMat, cullMng);

switch (precullResult){
        case VA_PRECULLRESULT_CONT:
                        cullManagerContinueTraverse(cullMng, TRUE);
                        break;
        case VA_PRECULLRESULT_CONT_ALLIN:
                        cullManagerContinueTraverse(cullMng, TRUE);
                        cullManagerDisableCullingTestOnBranch( cullMng);
                        break;
        case VA_PRECULLRESULT_PRUNE:
                        cullManagerContinueTraverse(cullMng, FALSE);
                        break;
}
```

En el caso del *nodo vaSkeleton*, el objetivo principal de la función de *pre-cull* llamar a la función ***vaSklNodePreCull(),*** encargada comprobar si la distancia del actor a la cámara es mayor que la distancia de desactivación del *nodo vaSkeleton,* y actuar en tal caso evitando el procesado y dibujado del subgrafo

de escena que depende de él. Esta rutina retorna los mismos valores que la función *vaActNodePreCull(),* indicando la forma en la que se ha de seguir realizando el culling. Una representación genérica del código llamando en la función de *pre-cull* de un *nodo vaSkeleton* es la siguiente:

```
//Obtener el puntero a la estructura que proporciona información sobre el culling.
void *cullMng = sgGetCullManager()

//Obtener el puntero al nodo que esta llamando a esta función de callback.
void *sgNode = cullManagerGetCurNode(cullMng);

//LLamar a la rutina principal de gestion del nodoSkeleton.
int precullResult = vaSklNodePreCull( sgNode, cullMng);

switch (precullResult){
        case VA_PRECULLRESULT_CONT:
                        cullManagerContinueTraverse(cullMng, TRUE);
                        break;
        case VA_PRECULLRESULT_CONT_ALLIN:
                        cullManagerContinueTraverse(cullMng, TRUE);
                        cullManagerDisableCullingTestOnBranch( cullMng);
                        break;
        case VA_PRECULLRESULT_PRUNE:
                        cullManagerContinueTraverse(cullMng, FALSE);
                        break;
}
```

En el *nodo vaLod*, la misión principal de la rutina de pre-cull es llamar a la rutina *vaLodNodePreCull()*, la cual emplea internamente la función *vaLodSelectChild()* para seleccionar el nodo hijo más adecuado a esa distancia. La forma de esta función es la siguiente:

```
//Obtener el puntero a la estructura que proporciona información sobre el culling.
void *cullMng = sgGetCullManager()

//Obtener el puntero al nodo que esta llamando a esta función de callback.
void *sgNode = cullManagerGetCurNode(cullMng);

//LLamar a la rutina principal de gestion del nodo LOD.
vaLodNodePreCull( sgNode, cullMng);
```

Con respecto a las funciones de callback de *pre-draw* y *post-draw*, resultan mucho más simples que las de *pre-cull*, reduciéndose básicamente a realizar llamadas a las funciones *vaActNodePreDraw()*, *vaActNodePostDraw()*,*vaSklNodePreDraw() y vaSklNodePostDraw()*.

La funciones *vaActNodePreDraw()* y *vaActNodePostDraw()* se encargan principalmente de aplicar la matriz del *nodo Actor*, además, llaman internamente a las funciones asociadas al nodo Actor mediante las funciones *vaActclassSetDrawFuncs()* o *vaActorSetDrawFuncs(),* y se encargan de dibujar las *bspheres* del actor si es necesario. Las funciones *vaSklNodePreDraw()* y *vaSklNodePosDraw()* se encargan principalmente de aplicar la matriz de transformación del *nodo Skeleton*.

8.6 Arquitectura Modular.

Desde el punto de vista de implementación práctica de los contenidos de este trabajo, se realizan dos aportaciones principales:

- Se proporciona una librería que contiene todos los métodos y estructuras propuestos por este trabajo (descrita en los apartados anteriores).
- Se propone una arquitectura modular, basada en la utilización de dicha librería, y que contempla todos los aspectos de la integración de los actores virtuales en una aplicación de simulación.

El objetivo principal de la arquitectura modular propuesta, es lograr que el trabajo desarrollado con los actores virtuales sea independiente de la aplicación en la que se integre, consiguiendo de este modo, la implementación de actores virtuales reutilizables, así como de módulos de gestión de comportamiento también reutilizables. Esta arquitectura modular permitirá, además, la integración entre actores desarrollados por diferentes equipos de personas, o la utilización de módulos de gestión de comportamiento desarrollados de forma externa. Las estructuras y métodos propuestos en esta tesis actúan de forma efectiva como capa de bajo nivel para el soporte de actores virtuales, actuando como un substrato adecuado para la definición de capas de más alto nivel. La propuesta de arquitectura global para la gestión de actores virtuales es la mostrada en la siguiente figura:

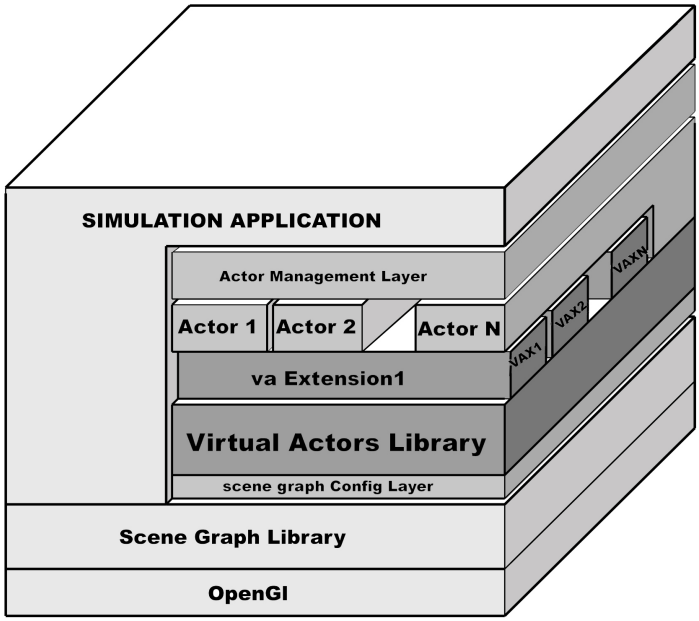


Figura 8-1. Organización modular de una aplicación de simulación empleando actores virtuales según la librería implementada.

Como se puede observar, todo el bloque encargado de la gestión de los actores virtuales, se encuentra enmarcado entre la aplicación total de simulación, y el grafo de escena empleado para el desarrollo del simulador. La capa etiquetada como "scene graph config layer" hace uso de las funciones descritas en "vaSgcfg.h", para adaptar la librería de actores al grafo de escena empleado. Por encima de la librería de actores se observan dos tipos de módulos: Módulos encargados de la definición de un tipo de actor virtual

(etiquetados como Actor1, Actor2, etc.), y módulos encargados de la definición de algún aspecto del comportamiento de los actores virtuales (etiquetados como vaExtension1, etc.). Un módulo de definición de un actor virtual puede emplear varios módulos de extensión, un módulo de extensión puede, de forma similar, ser empleado sobre distintas clases de actores.

El objetivo principal en el proceso de creación de una aplicación de simulación con actores virtuales es definir una o varias clases de actores. Cada una de estas clases de actores será definida como un módulo separado, y se accederá a ella mediante un "microAPI" de funciones que definen todas las acciones que pueden ser realizadas sobre ese tipo de actor. Así, por ejemplo, si en una aplicación de simulación se necesitase disponer de actores de tipo humano y de tipo ave, se crearían dos ficheros de nombres "humanVa.h" y "birdVa.h", dentro de estos ficheros habría funciones tales como *humanCreateNew()*, *humanWalkToPoint()*, *birdSetCurrentPosition()* o *birdLookToPoint()*. La implementación de dichas funciones se encontraría dentro de unos ficheros de nombres "humanVa.c" y "birdVa.c".

En el proceso de definición de una clase de actores virtuales, es habitual que se necesite, además, el empleo de varias extensiones, así puede ser necesaria la existencia de una extensión de gestión de posturas, de keyframing, un mecanismo de control de mirada, o un mecanismo de control del desplazamiento. Todas estas capacidades adicionales han de ser implementadas mediante el mecanismo de extensión proporcionado por la librería de actores virtuales. Cada uno de los módulos de extensión es almacenada en un módulo separado, definido mediante un fichero .c y un fichero .h de cabecera. Para evidenciar que estos ficheros constituyen una extensión de la librería de gestión de actores se les asigna un nombre que comienza con el prefijo "vax", así un módulo de gestión de posturas estaría formado por los ficheros "vaxPose.c" y "vaxPose.h", o un módulo de control del parpadeo por los ficheros "vaxBlink.c" y "vaxBlink.h". Las extensiones no son dependientes de una clase de actor en particular, así por ejemplo, la extensión de control del parpadeo, podrá ser empleada por todas las clases de actores que tengan párpados. En algunas ocasiones, un módulo de extensión tiene que ser configurado para trabajar con cada clase de actor, este tipo de configuración suele ser muy sencilla: En el caso del mecanismo de parpadeo, bastaría con indicar cuales son los índices de los cuatro grados de libertad correspondientes con la rotación de los párpados superiores e inferiores. También es posible crear unos módulos de extensión empleen internamente a otros, así, por ejemplo, el módulo de parpadeo puede emplear internamente al módulo de posturas, (definiendo una postura que coloca los párpados en posición cerrada, y otra que los coloca en posición abierta). Esta capacidad de organización jerárquica de los módulos de extensión proporciona un método muy potente para la descripción de actores con comportamientos muy complejos, y puede ser observada en más detalle en el ejemplo de aplicación mostrado en el siguiente capítulo.

Por último la capa nombrada como "Actor Managemen Layer" se encarga de crear, configurar y definir el comportamiento de los actores, recurriendo para ello, a la utilización de los "microAPIs" proporcionados por los módulos de definición de actores descritos anteriormente. En el siguiente capítulo se mostrará un ejemplo práctico de la utilización de esta arquitectura, para la integración de un gran número de actores virtuales sobre una escena generada con *OpenGl Performer*.

8.7 Fases del desarrollo de una aplicación con actores virtuales mediante la librería y la arquitectura modular propuestas.

La integración de actores virtuales dentro de una aplicación de simulación puede ser considerada como formada por dos grandes fases:

- Definir cada uno de los actores virtuales que van a ser necesarios en la aplicación, y proporcionar un pequeño API de funciones de muy alto nivel ("microAPI") con el que se pueda controlar las acciones de los actores dentro de la simulación.
- Definir un módulo que controle a todos los actores existentes en la simulación mediante la utilización de las funciones proporcionadas por sus "microAPIs". Este módulo es representado en la *Figura 8-1* bajo el nombre de "Actor Management Layer".

La definición de cada tipo de actor virtual comienza con el análisis de las características de los actores que van a intervenir en la escena de simulación, y finaliza con la obtención de un pequeño API que proporcione un control de muy alto nivel sobre cada uno de esos actores. El proceso total de definición de un tipo de actor virtual, según la librería y arquitectura propuestas, se puede considerar dividido en 6 fases:

1. Descripción de las características básicas.

- Descripción del aspecto externo del actor virtual.
- Descripción de la topología del actor virtual y de las acciones que llevará a cabo.

2. Estudio de los niveles de detalle

- Definición de los niveles de detalle topológicos.
- Asignación de los niveles de detalle geométricos, teniendo en la influencia de los topológicos.
- Asignación de los niveles de detalle comportamentales, teniendo también en cuenta los niveles de detalle topológicos.

3. Definición de la estructura articulada del actor.

- Creación de la estructura *vaActclass*.
- Creación de las estructuras *vaDof*. Asignación de nombres adecuados y límites.
- Creación de las estructuras *vaSkelprot*. Asignación de nombres adecuados, definición de offsets, distancias de desactivación y conexión con sus grados de libertad (estructuras *vaDof* definidas anteriormente).
- Definición de la estructura articulada del actor mediante la conexión entre las estructuras *vaSkelProt*, y también entre la estructura *vaActClass* y los *vaSkelprot* que actúan como padres de la estructura jerárquica. .
- Llamada a la función *vaActclassCompile()* que se encarga de generar las tablas indexadas para acceso a las estructuras *vaDof* y *vaSkelprot*. A partir de este momento se puede acceder de una

forma rápida a cualquier grado de libertad, o punto de articulación, mediante la utilización de un índice. El índice asignado a cada uno de estos elementos puede ser obtenido mediante las funciones *vaActclassFindSklIndexByName()* y *vaActclassFindDofIndexByName()*.

4. Definición del comportamiento del actor.

- Análisis de los distintos módulos de gestión de comportamiento necesarios. Toma de decisiones sobre cuales se han de implementar como una extensión (criterios de complejidad, reutilización, etc.), o directamente en el interior de la clase de Actor.
- Definición de los módulos de comportamiento externos definidos como objetos de tipo *vaExtension*.
- Definición de los módulos de comportamiento internos, y almacenamiento dentro de los datos de usuario de la estructura *vaActclass*.
- Configuración del *vaActclass* para uso de cada extensión.

5.Vinculación de la geometría con la estructura articulada.

La asignación de la geometría a la estructura articulada del actor puede ser realizada de varios modos, siendo el más sencillo la definición de un actor prototipo. Una vez definida la estructura topológica de un *vaActclass* siguiendo los pasos anteriores, la definición de la estructura básica de un actor en el grafo de escena mediante la creación de su nodo *vaActor* y sus correspondientes nodos *vaSkeleton,* es realizada directamente mediante la orden la orden *vaActorNew().* Una vez definida la estructura de nodos *vaActor* y *vaSkeleton*, se ha de proceder a la adición de los nodos *vaLod,* y de los nodos con información geométrica que definen las diferentes partes. El resto de los actores de esta clase serán copias de este actor prototipo generadas mediante la orden *vaActorClone()*. Esta forma básica de creación de actores puede ser refinada de varios modos, ya citados anteriormente, tales como la utilización de un único actor prototipo con varias texturas alternativas, la utilización de varios actores prototipo, la utilización de un conjunto de piezas intercambiables, el empleo de modelos parametrizados, etc.

6.Definicióndel "microAPI" del actor.

Es el paso final, consiste en definir un conjunto de ordenes de muy alto nivel que ocultan totalmente la complejidad de las capas intermedias que han sido necesarias. Mediante este "microAPI", el desarrollador de la aplicación de simulación puede controlar a los actores virtuales de una forma muy cómoda, olvidándose de la complejidad de las capas inferiores.

Los detalles internos de cada una de estas fases pueden ser apreciadas en detalle en el ejemplo de aplicación presentado en el siguiente capítulo.

8.8 Conclusiones.

Con el objeto de demostrar la utilidad práctica de los resultados de este trabajo, se ha desarrollado una librería de 134 funciones que proporciona acceso a las estructuras y métodos descritos, y, además, se ha propuesto una arquitectura modular que contempla todos los aspectos de la integración de actores virtuales en una aplicación de simulación.

La librería de gestión de actores ha sido desarrollada siguiendo una metodología ingeniería del software basada en la orientación a objetos, y el resultado final ha sido una librería en C que proporciona acceso a las 10 clases empleadas en el modelo de objetos. Los prototipos de dichos funciones han sido definidos en dos ficheros de cabecera: "vaApi.h", que proporciona acceso a las clases principales, y "vaSgcfg.h" encargado de independizar a la librería, del grado de escena. Se han presentado todas las funciones empleadas por la librería, prestando especial atención a detallar la forma en la que se consigue que la librería sea independiente del grafo de escena, describiendo los requisitos mínimos de un grafo de escena para poder ser empleado con esta librería, y también, la forma en la que la gestión de los actores es integrada dentro de procesado propio del grafo de escena.

La arquitectura modular propuesta, aborda el problema de la integración de los actores en la aplicación de simulación desde un punto de vista global, yendo desde la definición de una capa que hace a los actores independientes del grafo de escena, hasta la definición de módulos de alto nivel encargados de la gestión del comportamiento, y la utilización de dichos módulos para la definición de actores que proporcionen un API de muy alto nivel.

Desde el punto de vista del integrador, el trabajo queda dividido en dos grandes bloques:

- Crear actores que presentan cierto nivel de autonomía (es decir, actores que sean directamente controlables a través de un "microAPI" de alto nivel).
- Integrar los actores en la escena, e indicarles las actividades que tienen que realizar (empleando para ello los "microAPI" definidos anteriormente).

Los pasos que van desde el análisis del tipo de actor que se necesita, hasta la generación de su "microAPI" han sido descritos a grandes rasgos en este capítulo, y serán mostrados en detalle mediante el ejemplo de aplicación presentado en el siguiente.

Capítulo 9. Ejemplo de Aplicación.

9.1 Índice.

9.2 Introducción.

Este capítulo contiene un ejemplo de utilización de las estructuras y métodos propuesto en esta tesis. La librería y arquitectura descritas en el capítulo anterior serán empleadas para integrar una gran cantidad de actores virtuales sobre aplicación de simulación ya existente.

La aplicación base sobre la que realizará la integración es la utilidad de visualización de escenas conocida como "*perfly*" proporcionada por la librería gráfica "*OpenGl Performer*". Dicha utilidad será empleada para visualizar una escena de simulación típica conocida como "*Performer Town*" consistente en una ciudad pequeña rodeada de campos y montañas. La librería y arquitectura propuestas serán empleadas para crear y controlar un conjunto elevado de aves que desarrollaran diversas actividades sobre el cielo de dicha ciudad.

En este prototipo de aplicación se emplearan la gran mayoría de las funciones de la librería de actores, constituyendo un ejemplo suficientemente significativo para estimar la fiabilidad de la librería.

El desarrollo del ejemplo de aplicación estará formado por las dos grandes fases descritas en el capítulo anterior: Creación de los actores virtuales con el objetivo final de obtener un "*microAPI*" que permita manejarlos mediante de comandos de muy alto nivel, y definición del módulo que integra a esos actores en la escena global controlando que tipo de actividades han de realizar. Este capítulo muestra de forma pormenorizada todos los detalles de ambas fases.

En los siguientes apartados se detallan cuales han sido los criterios seguidos para la elección de este ejemplo de aplicación, se proporciona una visión global de los distintos módulos necesarios para la implementación, se presenta primera fase del desarrollo, consistente en la definición de la clase de actor a emplear, y la segunda, consistente en la integración de actores de este tipo en la aplicación global. Se continúa con un apartado de observaciones sobre los resultados obtenidos en el ejemplo de aplicación, y se finaliza con un apartado de conclusiones sobre los resultados prácticos obtenidos.

9.3 Criterios para la elección del ejemplo de aplicación.

9.3.1 Criterios para la selección del grafo de escena y la aplicación de simulación base.

Respecto al grafo de escena, se ha decidido emplear el proporcionado por la librería *OpenGl Performer*, librería que ha sido el referente durante muchos años en el desarrollo de sistemas profesionales de simulación visual.

Respecto la aplicación sobre la que se integrarán los actores virtuales, se ha seleccionado la aplicación de visualización de bases de datos de *OpenGl Performer* conocida como "*perfly*", la cual ha constituido durante muchos años el punto de partida en el aprendizaje de la utilización de esta librería. Como escenario estático sobre el que los actores realizaran sus actividades se ha seleccionado el conocido como "*Performer Town*" (ver *Figura 9-1*), el ejemplo más conocido de utilización de esta librería.

Figura 9-1. Imagen de la escena "*Performer Town*" y su grafo de escena.

La "*Performer Town*" consiste en una representación de una pequeña ciudad distribuida en área cuadrada de aproximadamente 5 Kilómetros de lado. Esta ciudad está definida de modo que constituye un perfecto ejemplo para la demostración de los algoritmos de gestión de *culling* y nivel de detalle propios de un grafo de escena.

La utilidad "*perfly*" permite cargar distintos tipos de ficheros con información geométrica, y proporciona diferentes modos para modificar la posición de la cámara, disponiendo, por ejemplo, de una interfaz que permite caminar o volar sobre una determinada escena, también proporciona distintos tipos de controles sobre las condiciones atmosféricas o la iluminación de la escena. También presenta elementos de interfaz que permiten testear la forma en la que están funcionando los algoritmos de gestión de nivel de detalle y culling, en concreto permite aplicar un factor multiplicador al algoritmo de selección de LOD y así forzar

artificialmente la visualización del nivel de deseado, y también un modo en el que la pirámide con la que se comprueba el culling es forzada a ser menor permitiendo de este modo comprobar la forma en la que el culling está actuando. Por último presenta una serie de controles que poder conocer la información sobre el rendimiento de la aplicación en cada instante.

9.3.2 Criterios para la elección del tipo de actores a añadir sobre esa aplicación.

Sobre la escena de simulación anterior, consistente en una ciudad rodeada de varias granjas, sería interesante implementar varias clases de actores realizando diversos tipos de actividades: humanos paseando por las calles, animales de compañía como gatos o perros paseando con sus amos o jugando en alguna plaza, también en las granjas podría haber distintos tipos de animales, como grupos de vacas o caballos pastando en los prados, por último, también resultaría adecuado mostrar un cielo en el que vuelan distintas especies de aves. Si bien el objetivo de esta tesis hacer que la creación de este tipo de escenas sea posible, por razones de tiempo y espacio vamos a simplificar al máximo el problema, e implementar solamente el último tipo de actores, es decir, a un grupo de aves volando sobre la ciudad, Una vez concluida la lectura de este capítulo se tendrá una idea más clara de como las estructuras y métodos propuestos en este trabajo posibilitan la creación de este tipo de escenas con multitud de actores virtuales. El actor que se mostrará en la escena es el mostrado en las siguientes imágenes:

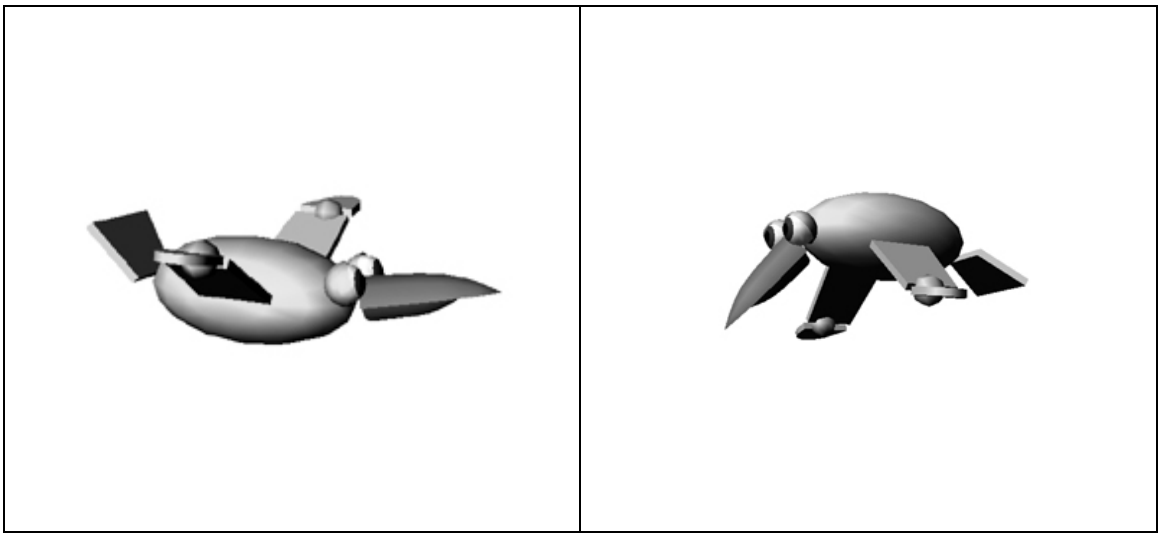


Figura 9-2. Actor virtual que será empleado en el ejemplo de aplicación.

Se puede pensar que hubiese sido más adecuada la implementación de una actor humano, pero hay varias razones para la elección de un actor más sencillo, la primera de ellas es que tanto su estructura jerárquica como la gestión de su comportamiento serían mucho más complicada, y su explicación dentro de este capítulo resultaría más difícil y bastante más extensa (por ejemplo, la implementación de un mecanismo que controle el desplazamiento de un humano sobre un suelo irregular, es más complejo que la implementación de un mecanismo que controle el vuelo de un ave), y la segunda es que se pretende demostrar que con las estructuras y métodos de propuestos en este trabajo se puede implementar cualquier tipo de actor virtual, no solamente humanos.

9.4 Organización modular del código.

Para la implementación del subsistema de gestión de actores dentro de la aplicación global de simulación va a ser necesario programar 8 módulos, cada uno de ellos implementado mediante un fichero .c y un su correspondiente fichero .h de cabecera. Dichos módulos pueden ser divididos en tres grupos.

1. Módulos encargados de la definición de la clase de actor. Pueden a su vez ser clasificados en dos subgrupos:
   - Módulos encargados de la definición de aspectos del comportamiento. Ficheros "*vaxLook.h*", "*vaxPose.h*", "*vaxBlink.h*", "*vaxKftable.h*" y "*vaxFly.h*". Todos ellos comienzan con el prefijo "*vax*" para indicar que se trata de módulos de extensión.
   - Módulo encargado de la definición y gestión del actor virtual. El fichero de cabecera de este módulo tiene el nombre "*birdVa.h*". Su nombre intenta reflejar que se trata de la implementación de la clase de actor "*Bird*" empleando la librería de actores virtuales (sufijo "*va*"). Este módulo contiene el "*microAPI*" de funciones de alto nivel que permite que un integrador pueda incorporar este tipo de actores virtuales en su aplicación con un esfuerzo mínimo. Una aplicación más compleja podría necesitar de los módulos "*humanVa.h*", "*dogVa.h*", "*horseVa.h*", etc.
2. Módulo encargado de gestionar la integración de los actores virtuales en la aplicación final, y su control mediante accesos a sus "microAPIs". El fichero de cabecera de dicho módulo tiene como nombre "birdsApp.h", y controla todo lo que ocurre con los actores de la escena de una forma independiente de plataforma.
3. Módulo de configuración de la librería para su utilización con *OpenGl Performer* (fichero vaInPf.h).

Las dependencias entre los distintos módulos definen la aplicación global son mostradas en diagrama mostrado en la *Figura 9-3*:

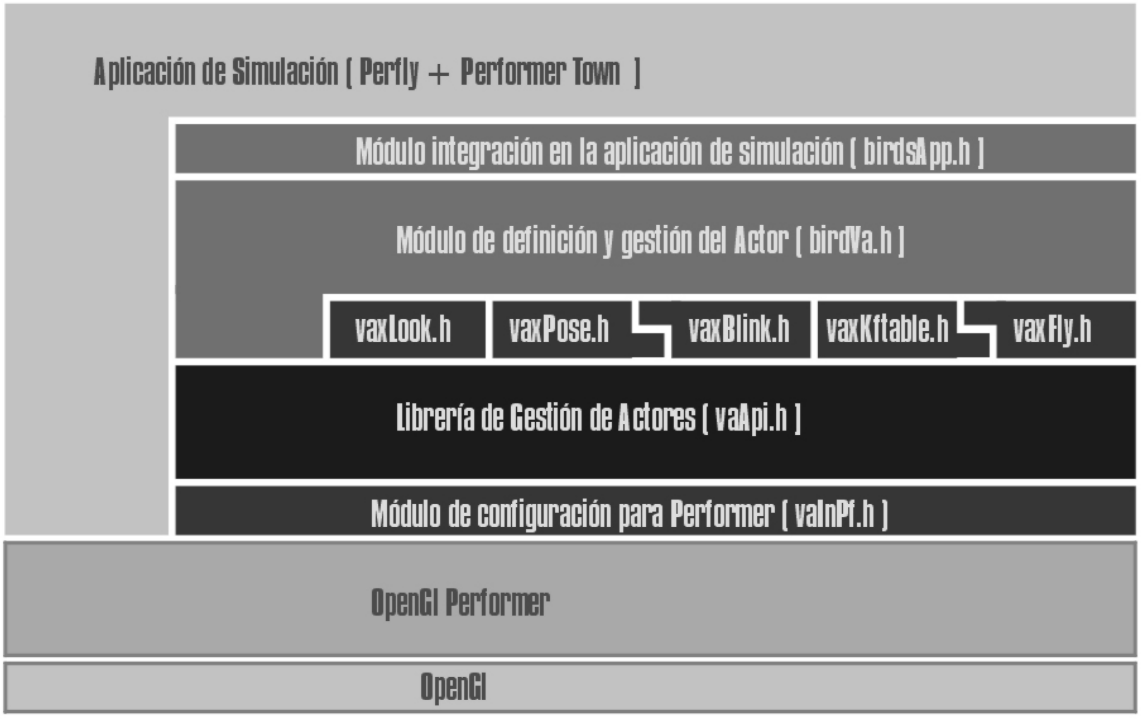


Figura 9-3. Organización modular del ejemplo de aplicación.

Se puede observar el módulo que representa a la librería de actores ("*vaApi.h*"), e inmediatamente debajo el módulo cuyas funciones de cabecera se encuentran en el fichero "*vaInPf.h*" encargado de configurar a la librería de actores para el trabajo sobre el grafo de escena *OpenGl Performer*, empleando para ello las funciones del API de actores definidas en el fichero "*vaSgCfg.h*". En la parte superior del módulo que representa a la librería de actores se pueden observar 5 módulos de extensión que serán empleados para implementar los mecanismos principales de vuelo, control de mirada y parpadeo (ficheros de cabecera "*vaxLook.h*", "*vaxBlink.h*" y "*vaxFly.h*"), y sus extensiones auxiliares de definición de poses y tablas de *Keyframing* ("*vaxPose.h*" y "*vaxKftable.h*"). El solapamiento existente entre los el módulo "*vaxBlink.h*" y "*vaxPose.h*" indica que el módulo de definición de poses proporciona un soporte de bajo nivel para la implementación del módulo de parpadeo. El actor virtual será definido empleando las funciones contenidas en "*vaApi.h*", y también en los ficheros de extensión. El "*microAPI*" de funciones de alto nivel que permitirán controlar a los actores de tipo "*Bird*" estarán definidas en el fichero de cabecera "*birdVa.h*". Por último, y con el fin de crear un ejemplo de aplicación que pueda ser rápidamente adaptado a otro tipo de grafo de escena, se ha definido un módulo que hace de interface entre la aplicación final y los actores virtuales. Este módulo, cuyo fichero de cabecera es "*birdsApp.h*", se encarga básicamente de traducir las ordenes enviadas por la aplicación de simulación a actuaciones sobre los actores virtuales.

A lo largo de este capítulo se irá mostrando el contenido de los distintos módulos.

9.5 Creación del actor virtual ejemplo.

En la creación del actor virtual ejemplo se emplearán las funciones del API de la librería de gestión de actores virtuales existentes en el fichero "*vaApi.h*" presentado en el capítulo anterior. En la creación de este actor virtual se mostrará en detalle la forma en la que se ha de definir una estructura de tipo *vaActclass,* y también se verán ejemplos prácticos de como se han de implementar e integran varias estructuras de tipo *vaExtension*. El resultado final será la definición de un "*microAPI*" de alto nivel que proporcionará un método muy sencillo de añadir actores de este tipo a una aplicación de simulación y controlar su comportamiento.

Este apartado comienza mostrando en primer lugar las características básicas del actor virtual, seguida de una visión detallada de sus niveles de detalle topológicos y sus niveles de detalle geométricos. A continuación se muestra la forma en la que han sido implementados los diferentes módulos de extensión, y, por último, se muestra el módulo que contiene todo el código de definición de esa clase de actor virtual y su comportamiento.

9.5.1 Características básicas.

En este apartado se presenta al actor virtual desde un punto de vista muy esquemático, describiendo su estructura jerárquica, y los nombres dados a sus puntos de articulación y a sus grados de libertad, también se describen a grandes rasgos las acciones que llevará a cabo este tipo de actor, y la forma en la que serán implementadas.

9.5.1.1 Descripción de la topología del actor virtual.

El actor virtual de tipo ave a implementar estará formado por 13 puntos de articulación, mediante los cuales será posible controlar la orientación de la mirada, la expresión facial, la apertura de la boca, la posición de la cola, y la posición de las alas.

En la *Figura 9-4*, mostrada a continuación, se puede observar una imagen del actor virtual, y una representación de su estructura jerárquica en la que se muestran los centros de cada *nodo vaSkeleton* y sus nombres.

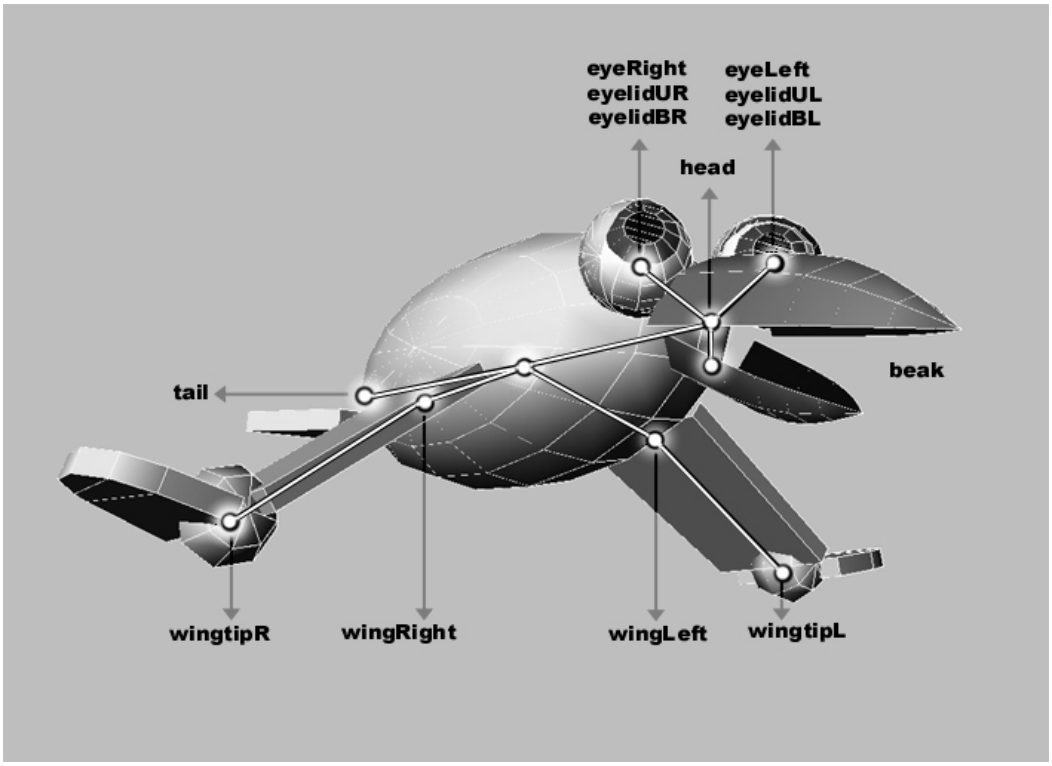


Figura 9-4. Estructura topólogica del actor y nombres de sus puntos de articulación.

Como se puede observar, los centros de los *nodos vaSkeleton* correspondientes los puntos de articulación del ojo y sus párpados superior e inferior coinciden en el mismo punto, sin embargo, como se puede observar más claramente en el grafo de escena representado a continuación (*Figura 9-5*), son nodos totalmente independientes.

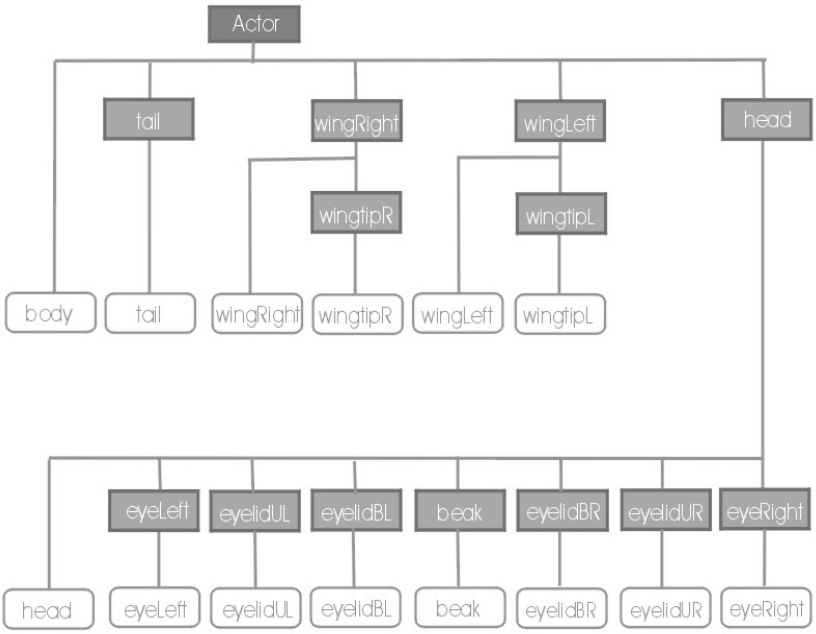


Figura 9-5. Representación de nodos *Actor* y *Skeleton* que definen al actor virtual.

En la figura anterior se puede observar una versión simplificada del grafo de escena que representará al actor virtual. Se puede observar como los nodos *wingtipR* y *wingtipL* dependen, respectivamente, de los nodos *wingRight* y *wingLeft*, y como los nodos de los párpados, ojos y pico dependen del nodo *Head*. En esta representación simplificada son necesarios trece nodos de tipo geométrico, cada uno de ellos dependiente de un *nodo Skeleton,* y una adicional (de nombre *"body"*) que dependería directamente del *nodo Actor*.

El grafo de escena que representa a este actor virtual se hará más complejo cuando se tengan en cuenta los niveles de niveles de detalle topológicos y geométricos, sin embargo, la topología de los *nodos Skeleton* permanecerá inalterable.

9.5.1.2 Definición de los parámetros de control.

La postura del actor será controlada a través de 19 grados de libertad, los nombres con los que estos grados de libertad serán identificados, y los nombres de los *nodos Skeleton* a los que pertenecen son mostrados en la siguiente tabla:

Nombre del nodo Skeleton	**Nombre del Grado de libertad (DOF).**
wingL	wingLElev
wingR	wingRtElev
wingtipL	wingtipRElev
wingtipR	wingtipLElev
tail	tailElev
head	headAzim
	headElev
	headTwist
eyeL	eyeLAzim
	eyeLElev
eyeR	eyeRElev
	eyeRElev
eyelidU	eyelidULOpen
	eyelidULExpr
eyelidUR	eyelidUROpen
	eyelidURExpr
eyelidBL	eyelidBLOpen
eyelidBR	eyelidBROpen
beak	beakOpen

Tabla 9-1. Listado de nombres asignados a los puntos de articulación y los grados de libertad.

9.5.1.3 Descripción de las acciones llevadas a cabo por el actor virtual.

El actor ha de estar preparado para realizar de forma automática una serie de actividades, y también ha de ser capaz de adaptar su comportamiento a los requerimientos de la aplicación final en la que sea integrado. La definición del comportamiento de un actor puede llegar a niveles de sofisticación muy elevados (por ejemplo la implementación de un sistema de visión artificial, o métodos de comportamiento basados en de inteligencia artificial), en este ejemplo se van a implementar tan sólo mecanismos de gestión de comportamiento de bajo nivel, pero son suficientemente ilustrativos como para mostrar la forma en la que se podrían implementar controles más complejos. Cada aspecto del comportamiento del actor será gestionado por un mecanismo de control específico. Los mecanismos de control que se implementaran sobre el actor ejemplo serán:

- Mecanismo de control del vuelo.
- Mecanismo de control de la mirada.
- Mecanismo de control del parpadeo.
- Mecanismo de control de la expresión facial.
- Mecanismo de control de la locución.

Algunos de estos mecanismos resultan muy sencillos y muy específicos de este tipo de actor, tal es el caso del mecanismo de control de la expresión facial, que actúa sobre los párpados superiores, o el mecanismo de control de la locución, que actúa sobre la parte inferior del pico. Dichos mecanismos serán implementados almacenando su información directamente en los datos de usuario de las estructuras *vaActclass* y *vaActor.*

Los mecanismos de control de vuelo, control de la mirada y control del parpadeo, resultan más complejos y una vez implementados, pueden ser reutilizados sobre otro tipo de actor virtual. En concreto el mecanismo de control de la mirada podría ser aplicado sobre cualquier tipo de actor que tenga cuello y dos ojos (por ejemplo un humano), el mecanismo de parpadeo sobre cualquier animal que tenga párpados superiores e inferiores, y el mecanismo de vuelo sobre cualquier ser que vuele. Por estas dos razones (complejidad y capacidad de reutilización), estos últimos mecanismos serán implementados empleando estructuras ***vaExtension***, y su información almacenada en los *extension slots* de las estructuras *vaActclass* y *vaActor*.

La forma en la que serán programados todos estos mecanismos será vista detalladamente en los últimos apartados de este capítulo, sin embargo, se va describir de forma abreviada cual es su forma de funcionamiento, y cuales son los grados de libertad sobre los que actúan:

Mecanismo de control del vuelo.

Define la posición del actor en la escena actuando sobre los valores de posición y orientación del *Reference Point* y el *Skeletons Root* de cada actor. También actúa sobre los grados de libertad de las alas y la cola, actuando sobre un total de 17 variables: las 6 variables que controlan la posición y orientación

del ***Referente Point*****,** las 6 que definen la posición y orientación del ***Skeletons Roo****t*, y, además, los grados de libertad ***wingLEelev***, ***wingRElev***, ***wingtipRElev***, ***wingtipLElev*** y ***tailElev***.

Mecanismo de control de la mirada.

Actúa sobre los grados de libertad de los ojos y el cuello para conseguir que el actor virtual mire a un punto determinado. En total controla 7 grados de libertad del actor:
headAzim, ***headElev***, ***headTwist***, ***eyeLeftAzim***, ***eyeLeftElev***, ***eyeRightAzim*** y ***eyeRightElev***.

Mecanismo de control del parpadeo.

Este mecanismo actúa sobre grados de libertad de los párpados superiores e inferiores (***eyelidULt***, ***eyelidUR***, ***eyelidBL***, ***eyelidBR***), forzándolos a que se cierren periódicamente. En total actúa sobre 4 grados de libertad: ***eyelidULOpen***, ***eyelidBLOpen***, ***eyelidUROpen*** y ***eyelidDROpen***.

Mecanismo de control de la expresión facial .

El mecanismo de control de la expresión facial es muy sencillo, actuando tan sólo sobre los párpados superiores, es decir, sobre los nodos *Skeleton* ***eyelidUL*** y ***eyelidUR***. Tan sólo emplea dos variables que controlan los grados de libertad: ***eyelidULExpr*** y ***eyelidURExpr***.

Mecanismo de control de la locución.

Es un mecanismo que actúa sobre el movimiento la parte inferior del pico (nodo *Skeleton* ***beak***), y que puede ser empleado para simular la emisión de sonidos. Tan sólo actúa sobre el grado de libertad ***beakOpen***.

9.5.2 Niveles de detalle topológicos.

Por la forma en la que esta construida el actor virtual y la importancia relativa de sus diferentes partes que lo forman, se puede observar claramente que existirá una distancia en la cual será imposible apreciar los movimientos realizados con los ojos, y otra distancia mayor a partir de la cual incluso el movimiento de la cabeza será de difícil percepción. Basándose en esto, y para no consumir recursos de cálculo de forma innecesaria, se han de fijara unas distancias a partir de los cuales los *nodos Skeleton* han de dejar de ser procesados:

- A los *20 metros* se desactivan los *nodos Skeleton* ***eyeL***, ***eyeR***, ***eyelidUR***, ***eyelidUL***, ***eyelidBR*** y ***eyelidBL***.
- A los *100 metros* se desactivan, además, los *nodos Skeleton* ***wingtipL***, ***wingtipR***, ***beak*** y ***tail***.
- Por último, a los *500 metros* se desactivan los *nodos Skeleton* ***wingL***, ***wingR*** y ***head***.

Representando estos cambios sobre la topología del actor se obtiene la siguiente tabla, en la que se pueden observar claramente los 4 niveles de detalle topológicos generados, y como afecta al número total de *nodos Skeleton* y grados de libertad.

Partes del cuerpo	**LOD 1 (0 a 20 Metros)**	**LOD2 (20 a 100 m)**	**LOD3 (100 a 500 m)**	**LOD5 (más de 500 m)**
eyelidsB (2)	2	0	0	0
eyes (2)	4	0	0	0
eyelidsU (2)	4	0	0	0
wingtips (2)	2	2	0	0
beak (1)	1	1	0	0
tail (1)	2	2	0	0
wings (2)	2	2	2	0
head (1)	3	3	3	0
número Skeletons	13 Skeletons	7 Skeletons	3 Skeletons	0 Skeletons
número Dofs	20 DOFs	10 DOFs	5 DOFs	0 DOFs

Tabla 9-2. Cambio del nivel de detalle topológico con la distancia.

Como se puede observar, la existencia de niveles topológicos hace que el número de parámetros de control (*DOFs*) también se reduzcan con la distancia. De forma coherente, los mecanismos encargados de la gestión del comportamiento han de ser adaptados para que no realicen cálculos sobre parámetros que no van a ser empleados en la representación del actor.

El hecho de que los *nodos Skeleton* presenten distancias de desactivación también hace necesario que la geometría que desaparece por eliminación de un *nodo Skeleton* y su subgrafo dependiente, pase a ser representada en otro nodo geométrico. Así pues, el nivel de detalle topológico afecta a la forma en la que

se han de definir los niveles de detalle geométricos. En los siguientes apartados se muestran ambos tipos de influencia.

9.5.2.1 Influencia sobre el nivel de detalle geométrico.

Los cambios mínimos a realizar en el grafo de escena, para que el actor virtual pudiese ser representado de un modo coherente con los niveles de detalle topológicos fijados en el apartado anterior, serían los siguientes:

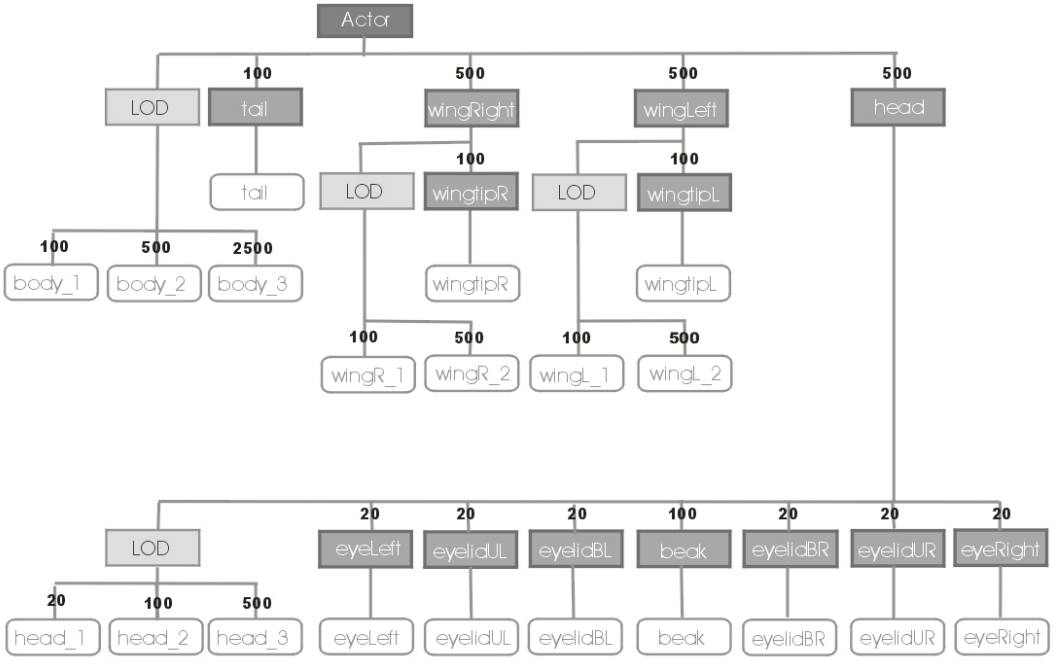


Figura 9-6. Grafo de escena mínimo en consideración a los niveles de detalle topológicos.

Como se puede observar, han aparecido diferentes nodos de tipo *LOD* que tienen unas distancias de cambio similares a las distancias de desactivación de los nodos *Skeleton*.

Los cambios producidos a nivel geométrico en el actor virtual, para que pueda ser representado de un modo coherente con los niveles de detalle topológicos descritos anteriormente, son los siguientes:

- A los 20 metros, los ojos y los párpados pasan a ser geometrías rígidas que ha de ser integrada en la geometría que representa la cabeza (geometría *head_2*).
- A los 100 metros, las puntas de las alas pasan a formar parte de la geometría de las alas (*wingR_2* y *wingL_2*), el pico inferior pasa a formar parte de la geometría de la cabeza (*head_3*), y la cola pasa a formar parte del cuerpo (*body_2*). Así pues, a los 100 metros solamente se emplean solamente 4 geometrías: la cabeza, las dos alas y el cuerpo.
- A los 500 metros los modelos de alas y cabeza quedan integrados en el modelo del cuerpo (*body_3*).

9.5.2.2 Influencia sobre el nivel de detalle comportamental.

Los mecanismos de gestión del comportamiento del actor actúan modificando las variables que definen la posición del actor en la escena y sus grados de libertad. El hecho de que el nivel de detalle topológico desactive ciertos nodos *Skeleton* con la distancia, hace que el número de variables a controlar por cada uno de estos mecanismos coherentemente también resulte reducido:

Influencia sobre el mecanismo de vuelo.

El mecanismo de gestión del vuelo del actor ejemplo ha de encargarse de la actualización de un total de 17 variables, sin embargo, la gestión de los niveles de detalle topológicos afecta del siguiente modo:

- A los 100 metros el actor desactiva los *nodos Skeleton* ***wingtipR***, ***wingtipL*** y ***tail*****,** con ello el mecanismo de vuelo puede dejar de controlar las variables correspondientes con los grados de libertad ***wingtipRElev*****,** ***wingtipLElev*** y ***tailElev***. Es decir, pasa a necesitar sólo 14 variables.

- A los 500 metros se desactivan, además, los *nodos Skeleton* ***wingR*** y ***wingL***, con lo cual deja de tener sentido el control de las variables ***wingLElev*** y ***wingRElev***, quedando reducido a 12 el número de variables a controlar por el mecanismo de vuelo.

Esto queda representado gráficamente en la siguiente tabla:

	0 m	100 m	500 m	∞
Núm. variables	18 variables	14 variables	12 variables	

Influencia sobre el mecanismo de control de la mirada.

El mecanismo de control de la mirada actúa sobre un total de 7 grados de libertad, viéndose afectado por los niveles de detalle topológicos del siguiente modo:

- A los 20 metros se desactivan los nodos *Skeleton* ***eyeL*** y ***eyeR***, y con ello, deja de ser necesario el control de los grados de libertad ***eyeLAzim***, ***eyeLElev***, ***eyeRtAzim*** y ***eyeRElev***, quedando el número de variables reducido a 3.

- A los 100 metros se desactiva, además, el nodo *Skeleton* ***head***, con lo cual se puede deshabitar totalmente el mecanismo de control de la mirada.

La siguiente tabla muestra como la distancia afecta al número de variables que controlan el mecanismo de mirada:

	0 m	20 m	100 m	∞
Núm. variables	7 variables	3 variables	0 variables	

Influencia sobre el mecanismo de control del parpadeo.

El mecanismo de control del parpadeo solamente se ve afectado por la desactivación de los *nodos Skeleton* de los cuatro párpados (***eyelidUL***, ***eyelidUR*****,** ***eyelidBR*** y ***eyelidDR***), lo cual ocurre a los 20 metros:

	0 m	20 m	∞
Núm. variables	2 variables	0 variables	

Influencia sobre el mecanismo de control de la expresión.

El mecanismo de control de la expresión solamente se ve afectado por la desactivación de los *nodos Skeleton* de los párpados superiores (***eyelidUL*** y ***eyelidUR***), que ocurre a los 20 metros:

	0 m	20 m	∞
Núm. variables	2 variables	0 variables	

Influencia sobre el mecanismo de control de la locución.

El mecanismo de control de la locución se ve afectado por la desactivación del *nodo Skeleton* correspondiente con la parte inferior del pico (***beak***), lo cual ocurre a los 100 metros:

	0 m	100 m	∞
Núm. variables	1 variables	0 variables	

9.5.3 Niveles de detalle geométricos.

Como hemos visto anteriormente, los niveles de detalle topológicos fuerzan a la existencia de niveles de detalle geométricos en determinadas distancias (observar *Figura 9-6*). Para la correcta representación del actor virtual es necesaria la existencia de niveles de detalle geométricos adicionales, que se activen sin relación con los cambios en la topología. La organización final de los niveles de detalle empleados en el actor virtual empleado como ejemplo, puede ser observada en el siguiente diagrama (*Figura 9-7*).

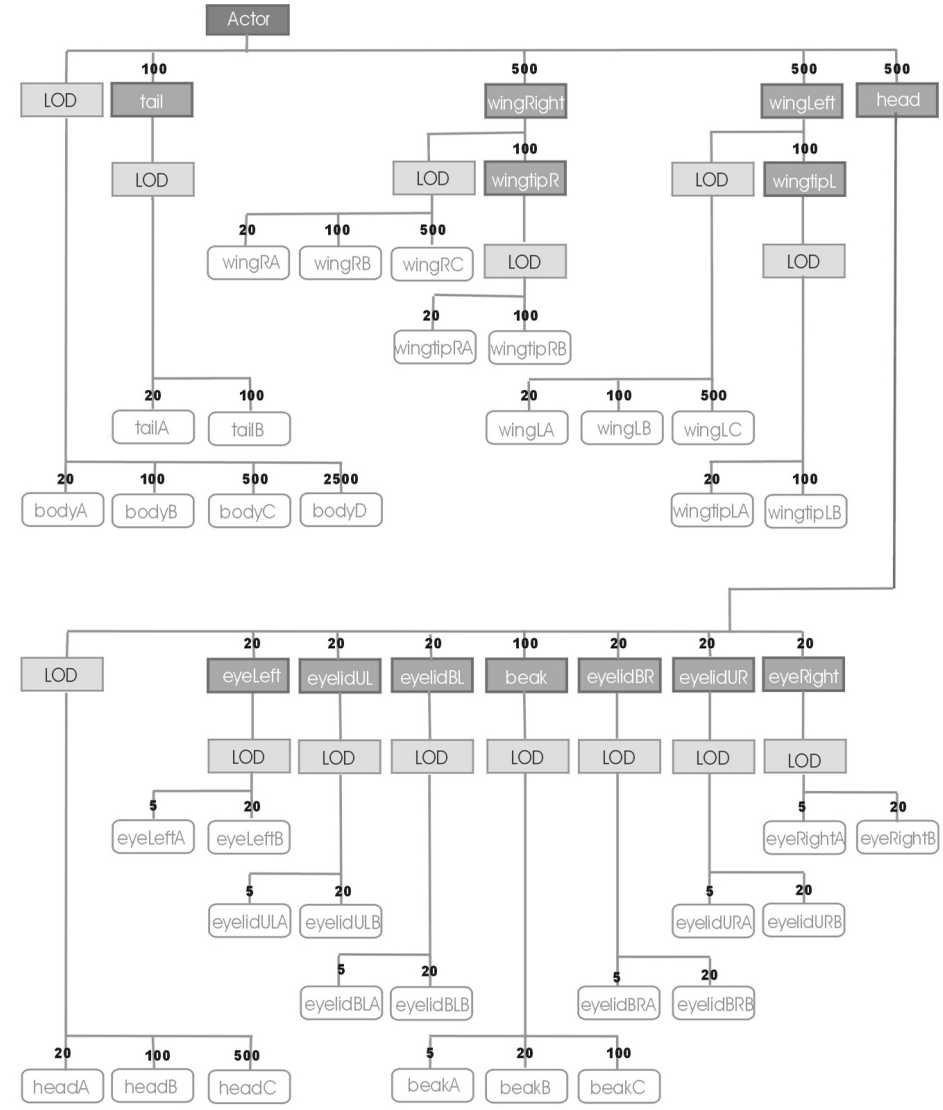


Figura 9-7. Grafo de escena final del actor virtual.

El cambio principal es que se ha añadido un nivel de detalle adicional para representar al actor en distancias muy cercanas a la cámara (menores de 5 metros), y también que se han añadido más variaciones en la geometría que representa a cada parte del actor, por ejemplo el pico es ahora representado con 3 geometrías alternativas.

En la siguiente tabla (*Tabla 9-3*), se muestran los cinco niveles de detalle geométricos del actor virtual ejemplo, para cada nivel de detalle se muestra el margen de distancias de distancias en el cual estará activo, y el número de triángulos empleado:

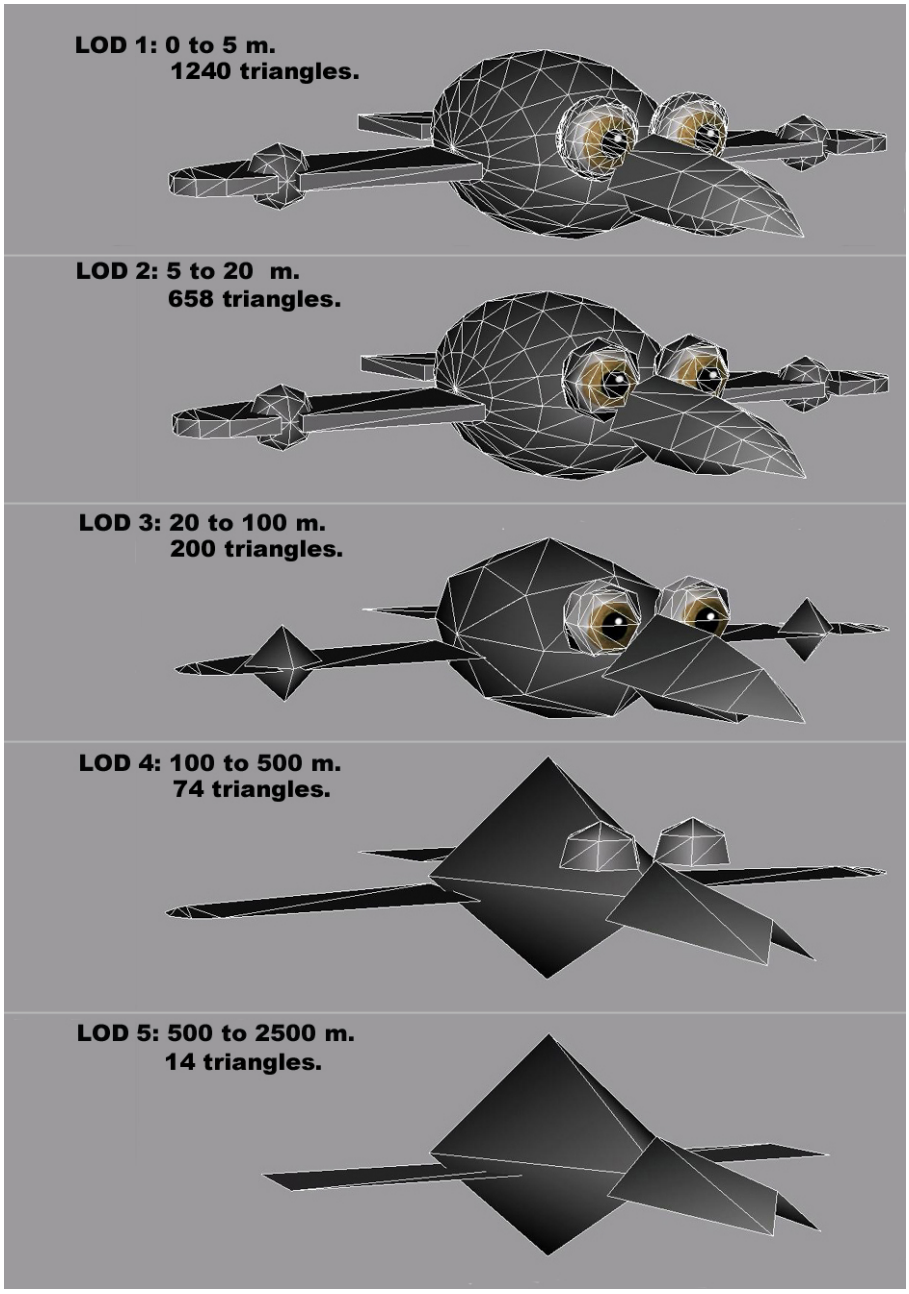


Tabla 9-3. Niveles de detalle geométricos globales del actor virtual ejemplo.

La tabla siguiente muestra los nombres de las distintas geometrías implicadas en la definición del actor virtual, y cuales son las distancias a las que están visibles:

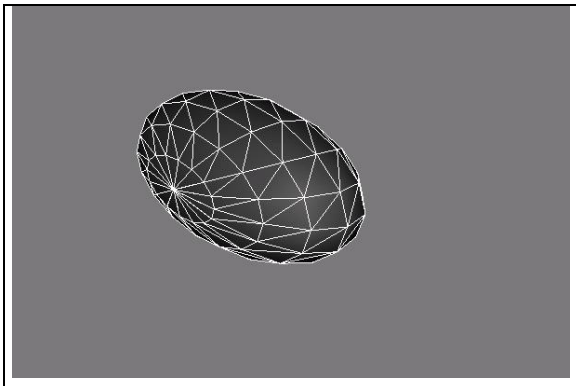
	0 m	5 m	20 m	100 m	500 m	2500 m
Partes	**0 a 5m**	**5 a 20 m**	**20 a 100 m**	**100 a 500 m**	**500 a 2500 m**	
body	bodyA		bodyB	bodyC	bodyD	
wings	wingA		wingB	wingC		
wingtips	wingtipA		wingtipB			
tail	tailA		tailB			
head	headA		headB	headC		
beak	beakA	beakB	beakC			
eyes	eyesA	eyesB				
eyelids	eyelidsA	eyelidsB				
num Triangles	1240	658	200	74	14	

Tabla 9-4. Variación de los nodos de geometría y el número total de vértices con la distancia.

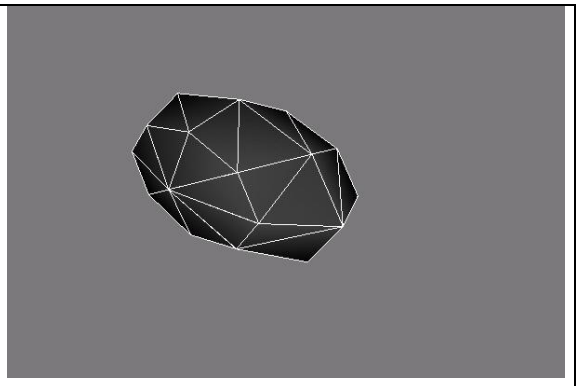
A continuación se muestra de forma detallada las distintas geometrías que componen el cuerpo del actor virtual, empleando los nombres usados en el grafo de escena mostrado en la *Figura 9-7*, e indicando cual es su cantidad de triángulos y los márgenes de distancia en el que aparecen visibles:

Niveles de detalle geométricos del cuerpo.

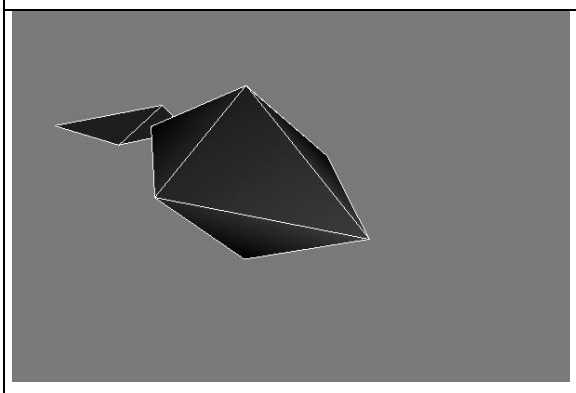
Las diferentes geometrías encargadas de representan la parte central del actor son las siguientes:



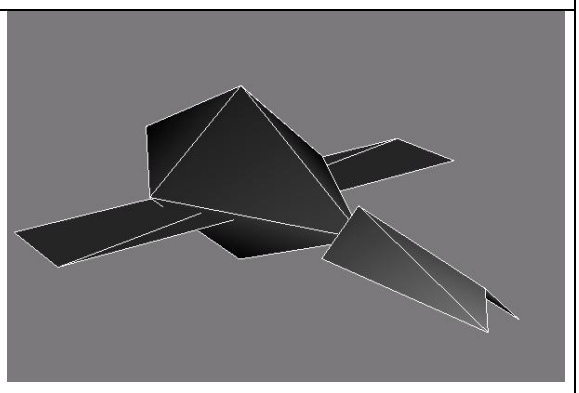
bodyA: de 0 a 20 metros. 224 triángulos.



bodyB: de 20 a 100 metros.48 triángulos.



bodyC: de 100 a 500 metros.10 triángulos.



bodyD: de 500 a 2500 metros.14 triángulos.

Tabla 9-5. Niveles de detalle geometricos del cuerpo.

Como se puede observar, la geometría ***bodyC*** incorpora la geometría de la cola, puesto que ésta deja de tener un comportamiento individual a los 100 metros debido a los niveles de detalle topológicos.

La geometría ***bodyD*** representa al actor completo a partir de los 500 metros, como se puede observar, lleva incluida una versión simplificada de las alas y de la cabeza. La cola ha sido eliminada por tener poca presencia visual a esas distancias.

Niveles de detalle geométricos de las alas.

Cada una de las dos alas del actor viene representada mediante tres niveles de detalle. En la siguiente tabla se muestran los correspondientes al ala izquierda:

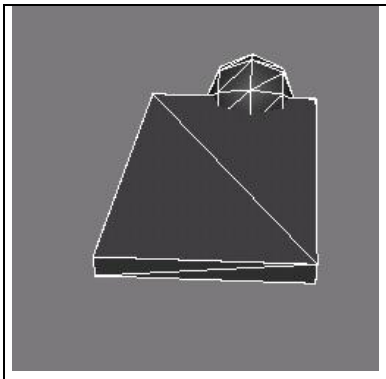
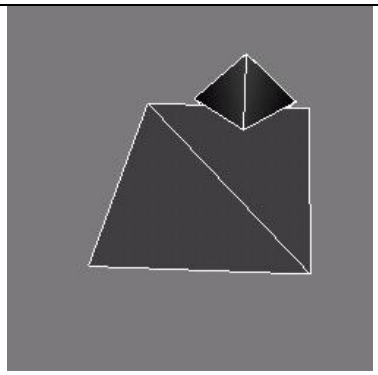
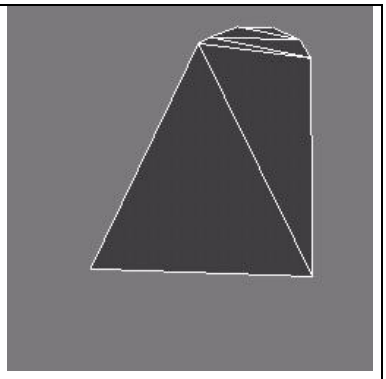
		
wingLA: de 0 a 20 metros. 60 triángulos.	**wingLB:** de 20 a 100 metros. 10 triángulos.	**wingLC:** de 100 a 500 metros. 6 triángulos.

Tabla 9-6. Niveles de detalle geométricos del ala.

Se puede observar que la geometría ***wingLC***, es un poco más larga de modo que representa también la punta del ala, eso es debido a que ésta desaparece por efecto del nivel de detalle topológico a los 100 metros.

A partir de los 500 metros las alas dejan de ser representadas como elementos individuales, y pasan a ser incorporadas a la geometría del cuerpo (***bodyD***).

Niveles de detalle geométricos de la punta del ala.

Cada una de las dos puntas de las alas del actor aparecen representadas mediante dos niveles de detalle. En la siguiente tabla se muestran los correspondientes a la punta del ala izquierda:

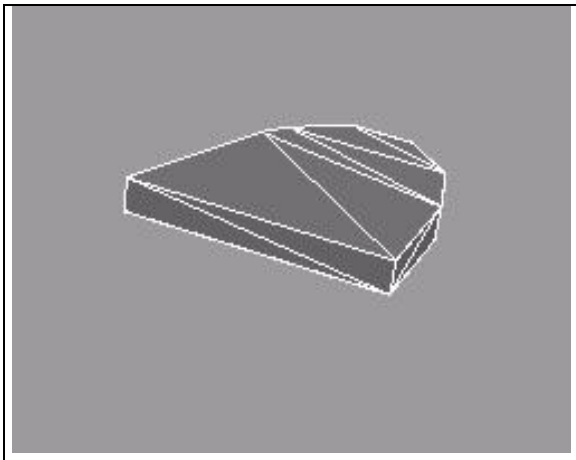
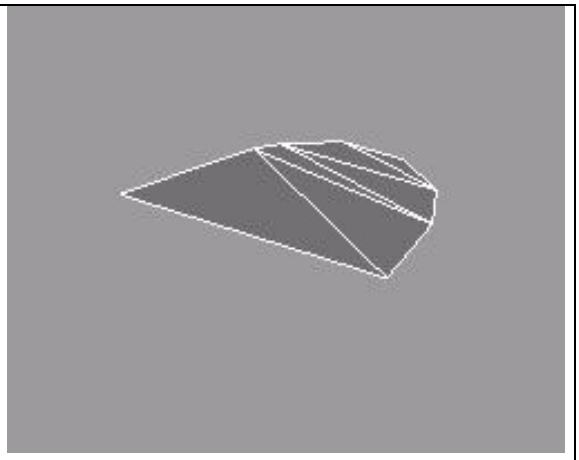
	
wingtipLA: de 0 a 20 metros. 28 triángulos.	**wingtipLB:** de 20 a 100 metros. 6 triángulos.

Tabla 9-7. Niveles de detalle geométricos de la punta del ala.

La punta del ala deja de ser representada como elemento individual a la distancia de 100 metros, pasando a ser incorporadas a la geometría del ala (***wingLC*** y ***wingRC***)

9.5.3.1 Niveles de detalle geométricos de la cola.

La cola del actor viene representada por los dos niveles de detalle mostrados a continuación:

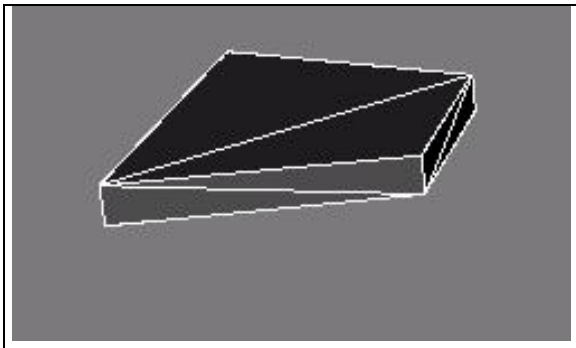
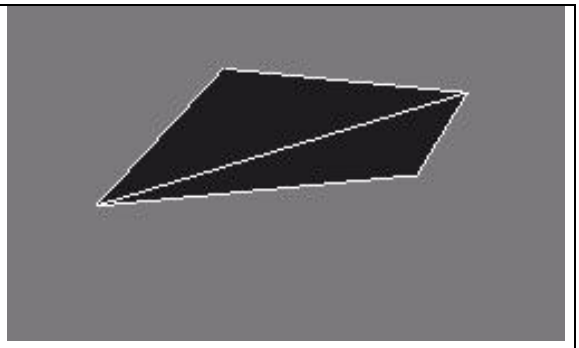
	
tailA: de 0 a 20 metros. 12 triángulos.	**tailB:** de 20 a 100. 2 triángulos.

Tabla 9-8. Niveles de detalle geométricos de la cola.

A partir de los 100 metros la cola deja de tener representación individual y pasa a ser integrada en la geometría del cuerpo (***bodyC***).

Niveles de detalle geométricos de la cabeza.

La geometría de la cabeza viene representada por 3 niveles de detalle, los cuales son mostrados en la siguiente tabla:

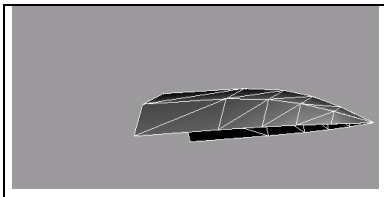
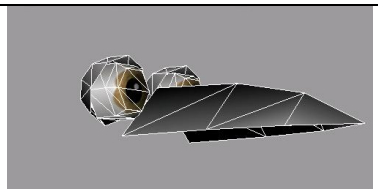
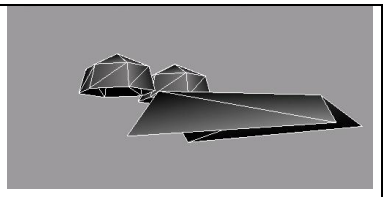
		
headA: de 0 a 20 metros. 36 triángulos.	**headB:** de 20 a 100 metros. 114 triángulos	**headC:** de 100 a 500 metros. 52 triángulos.

Tabla 9-9. Niveles de detalle geométricos del cabeza.

En distancias inferiores a los 20 metros, la cabeza viene solamente representada por el pico, puesto que el resto de los elementos (ojos, párpados y pico inferior), tienen representaciones geométricas separadas.

Se puede observar que a los 20 metros (modelo ***headB***), la cabeza pasa a tener una representación de los ojos y párpados del actor, esto es debido a que estas geometrías han dejado de ser representadas como elementos individuales a estas distancias. El modelo ***headB*** no representa los párpados inferiores (por tener poca importancia visual a esas distancias). El modelo ***headC*** es una versión más simplificada del *headB* en el cual se han eliminando las primitivas que representaban a los globos oculares, además de reducir el número de triángulos que representan al pico y los párpados.

A partir de los 500 metros, la cabeza deja de tener una representación individual, y pasa a ser absorbida por el modelo geométrico del cuerpo (***bodyD***).

Niveles de detalle geométricos de la parte inferior del pico.

La parte inferior del pico viene representada por los 3 niveles de detalle mostrados a continuación:

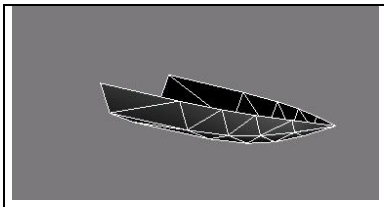
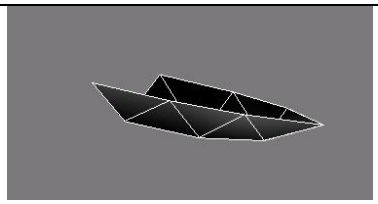
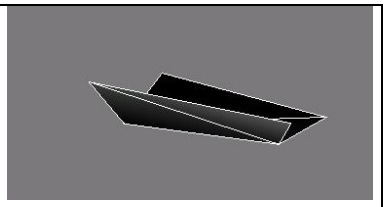
		
beakA: de 0 a 5 metros. 36 triángulos.	**beakB:** 5 a 20 metros. 10 triángulos	**beakC:** de 20 a 100 metros. 4 triángulos.

Tabla 9-10. Niveles de detalle geométricos de la parte inferior del pico.

En distancias superiores a los 100 metros el pico deja de ser representado por tener poca presencia visual.

Niveles de detalle geométricos del globo ocular.

Cada uno de los dos globos oculares del actor viene representado por los dos niveles de detalle mostrados a continuación:

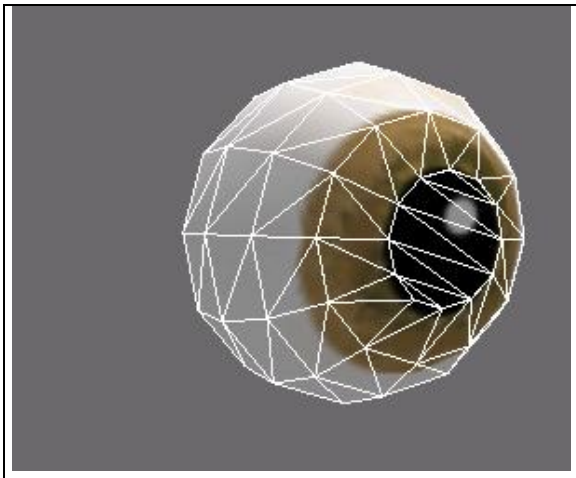
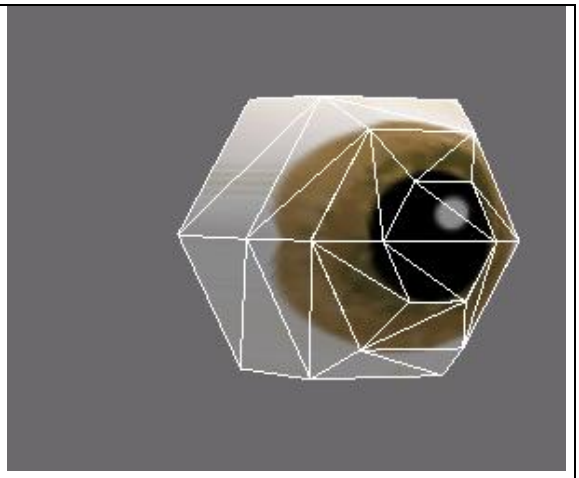
	
eyeLeftA: de 0 a 5 metros. 154 triángulos	**eyeLeftB:** de 5 a 20 metros. 52 triángulos.

Tabla 9-11. Niveles de detalle geométricos de un globo ocular.

En distancias superiores a los 20 metros los ojos dejan de ser representados como elementos individuales, y pasan a ser absorbidos por la geometría de la cabeza (***headB***).

Niveles de detalle geométricos de un párpado.

Cada uno de los cuatro párpados del actor virtual viene representados por los 2 niveles de detalle mostrados a continuación:

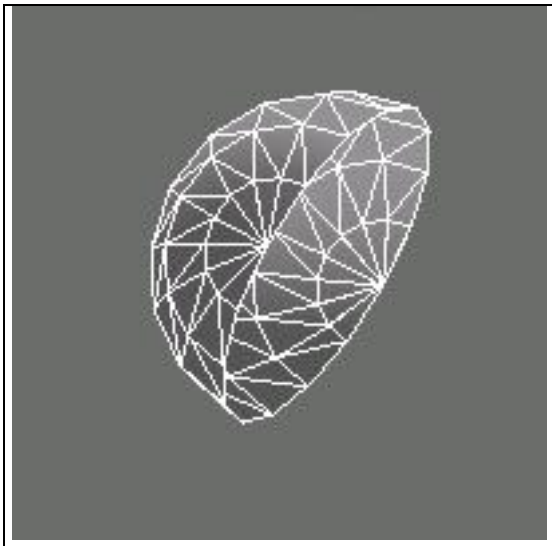
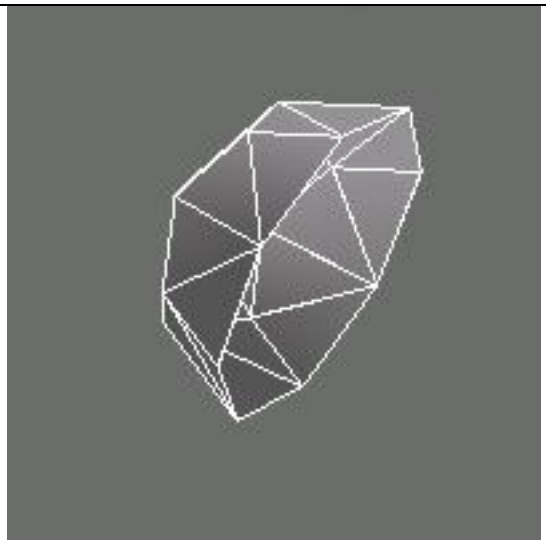
	
eyelidLA: de 0 a 5 metros. 112 triángulos.	**eyelidLB:** de 5 a 20 metros. 23 triángulos.

Tabla 9-12. Niveles de detalle geométricos de un EyeLid.

A partir de los 20 metros los párpados dejan de existir como elementos individuales, desapareciendo en el caso de los párpados inferiores, o pasando a ser integrados en la geometría de la cabeza (***headB***).

9.5.4 Implementación de los módulos de Extensión.

Como paso previo a la implementación del actor virtual se van a definir los módulos de gestión del comportamiento, implementados en forma de estructura *vaExtension*. Si bien, estos módulos de extensión han sido programados por primera vez para este trabajo y para este actor ejemplo, es de gran importancia destacar que su forma modular permite que sean reutilizadas en posteriores aplicaciones, y en actores virtuales totalmente diferentes. Así, por ejemplo, la extensión de control de dirección de mirada será empleada en este caso solamente al actor con forma de ave empleado en el ejemplo, pero podrá ser aplicada a cualquier tipo actor que tenga un cuello y dos ojos, bien sea un humano, un insecto o cualquier otro tipo de personaje.

La modularidad en la programación de las extensiones permite que puedan ser reutlizadas fácilmente, y, además, posibilita que puedan ser organizadas de forma jerárquica, existiendo extensiones de alto nivel que pueden estar actuando sobre otras extensiones de más bajo nivel. En este ejemplo se definirán un total de cinco extensiones: una de ellas encargada de definir y gestionar posturas estáticas en los actores virtuales (*pose Extension*), otra encargada de definir y aplicar secuencias de movimientos sobre un actor (*kftable Extension*), otra que controla el parpadeo del actor (*blink Extension*), otra que estará encarga de controlar la dirección de la mirada del actor (*look Extension*), y por último, una responsable de gestionar el vuelo del actor (*fly Extension*). Las extensiones de posturas y *keyframing* actúan como extensiones de bajo nivel destinadas a ser empleadas de forma auxiliar por otras de mayor nivel, así, la extensión de parpadeo emplea a la de posturas, y la de vuelo emplea a la de *keyframing*.

Cada extensión consta de un fichero con el código y un fichero de cabecera, así por ejemplo la extensión de gestión de posturas consta de un fichero de nombre ***"vaxPose.c"*** y otro de nombre ***"vaxPose.h".*** Las funciones exportadas por una extensión comienzan con el prefijo *"vax"*, así la función que se encarga de aplicar una postura ya definida sobre un actor, tiene el nombre ***vaxActorPoseApply()***.

Desde el punto de vista de implementación de la extensión, ésta siempre ha de constar de una función de inicialización, - la función *vaxPoseInit()* en el caso de la extensión de posturas -, y, además, varias funciones de extensión del *API* de actores. Estas funciones pueden afectar al *vaActorclass* o al *vaActor*. Así, en el caso de la extensión de posturas, la función *vaxActclassPoseCreate()* y *vaxActclassPoseDelete()* amplían el API de la *Actclass,* y la función *vaxActorPoseApply(),* amplia el API del *Actor*.

En los siguientes apartados se muestra el contenido de los ficheros .c y .h de las cinco extensiones creadas para este ejemplo. La primera de ellas (*vaxPose*) será explicada con más detalle puesto que las siguientes siguen un patrón similar.

9.5.4.1 Extensión de gestión de Posturas.

En el siguiente fragmento de código se muestra el contenido del fichero de cabecera *"vaxPose.h"*. Dicho fichero está formado por tres zonas, en la primera se encuentra la función de inicialización de la extensión, en la siguiente las funciones de ampliación del API del *Actorclass,* y en la tercera las funciones de ampliación del API del *Actor:*

```
#ifndef __VAX_POSE_H__
#define __VAX_POSE_H__
#include "vaApi.h"

//---- Initialization API --------------------------------------------------//
extern vaExtension *vaxPoseInit( );

//---- Actclass API --------------------------------------------------------//
extern int  vaxActclassPoseCreate( vaActclass *ac,  int numDofs,   int *dofIndexes, float *dofValues);
extern void vaxActclassPoseDelete( vaActclass *ac,  int poseIndex);

//---- Actor API -----------------------------------------------------------//
extern int vaxActorPoseApply(vaActor *act, int poseIndex, float transTime);
#endif
```

A continuación se muestra el contenido del fichero *"vaxPose.c"* que contiene el código de la extensión. El fichero aparece dividido en varios fragmentos para poder intercalar comentarios:

```
#include "stdlib.h"
#include <string.h>
#include "vaApi.h"
//---------Pose extension slot -------------------------------------------//
typedef struct poseGobalDataStr{
   int extIdentifier;
}poseGlobalData;
poseGlobalData *poseGdata;

typedef struct vaxActPoseStr{
   int               poseIndex;          // Indice de la pose.
   float             startTime;          // Instante de inicio de la transicion.
   float             transitionTime      // Duracion de la transicion.
   struct vaxActPoseStr *next;
}vaxActPose;

typedef struct vaxPoseStr{
   int       numDofs;
   int      *dofIndexes;                 // Indices de los dofs a los afecta
   float    *dofValues;                  // Valores que fija.
}vaxPose;

typedef struct poseActclassDataStr{
   int       numPoses;         // Numero de posturas predefinidas.
   vaxPose   **poses;          // Lista de posturas predefinidas.
}poseActclassData;

typedef struct poseActorDataStr{
   int       numCurPoses;
   vaxActPose *curPoses;
}poseActorData;
```

La primera pare del fichero define las estructuras de datos necesarias. La primera estructura de nombre ***poseGlobalData*** almacena únicamente un valor entero que actúa como identificador de la extensión. También se puede observar como existe un puntero de tipo global a ese tipo de estructura. En la fase de inicialización se solicita a la librería de actores que genera un identificador único que será empleado como identificador de la extensión, y también indicara el índice de los *extension slot* en el que será almacenada su información dentro de las estructuras *vaActclass*, *vaActor*, *vaSkelprot* y *vaDof.*

La extensión de gestión de posturas que se está tratando tan sólo almacena información en los *extension slots* del *vaActor* y de la *vaActorClass*, no usando, por tanto, ni la zona del *vaDof* ni la zona del *vaSkelprot*. La estructura ***poseActclassData*** es almacenada en el *extension slot* del *vaActclass,* y la estructura ***poseActorData*** es almacenada en el *extension slot* del *vaActor*.

La estructura ***vaxPose*** almacena una postura empleando para ello un array con los índices de los grados de libertad a los que afecta, y un array paralelo con los valores que dichos grados de libertad adoptan para esa postura. Como se puede observar, la estructura *poseActclassData* almacenará una lista de estructuras de este tipo, éstas serán todas las posturas predefinidas que el actor podrá adoptar.

El *actor* tendrá una lista de estructuras de tipo ***vaxActPose***, que, como se puede observar, consisten en un índice que identifica una pose en la lista de poses del *poseActclassData,* y también información sobre el momento en el que comenzó la transición hacia pose y la duración total de la transición. Así pues, un actor puede cambiar de una postura a otra de una forma progresiva, y puede estar ejecutando varios cambios de postura simultáneamente.

```
static void *poseActclassDataCreate(void *data){
   vaActclass *ac = (vaActclass *)data;
   poseActclassData *acData  = (poseActclassData *)vaSgMalloc(sizeof(poseActclassData));
   acData->numPoses    = 0;
   acData->poses          = NULL;
   return acData;
}

static void *poseActorDataCreate(void *data){
   vaActor *act = (vaActor *)data;
   poseActorData *actData    = (poseActorData*)vaSgMalloc(sizeof(poseActorData));
   actData->numCurPoses    = 0;
   actData->curPoses          = NULL;
   return actData;
}

static void poseActclassDataDelete(void *data){
   vaActclass *ac    = (vaActclass *)data;
   int slotIndex      = poseGetExtIdentifier();
   poseActclassData *acData = (poseActclassData *)vaActclassGetExtData(ac, slotIndex);
   for (int i = 0; i < acData->numPoses; i++)
        vaSgFree(acData->poses[i]);
   vaSgFree(acData->poses);
}
```

```
static void poseActorDataDelete(void *data){
   vaActor *act          = (vaActor *)data;
   int slotIndex         = poseGetExtIdentifier();
   poseActorData *actData = (poseActorData *)vaActorGetExtData(act, slotIndex);
   vaxActPose *ptr       = actData->curPoses;
   while (ptr != NULL){
        vaxActPose *auxPtr = ptr;
       ptr = ptr->next;
        vaSgFree(auxPtr);
        }
   vaSgFree(actData);
}

static void poseRemoveFromActor(vaActor *act, vaxActPose *actPose){
   int slotIndex = poseGetExtIdentifier();
   poseActorData *actData = (poseActorData *) vaActorGetExtData( act, slotIndex);
   if (actData->curPoses == actPose){
      actData->curPoses =  actPose->next;
      vaSgFree( actPose );
      actData->numCurPoses--;
   }
   else{
      vaxActPose *ptr = actData->curPoses;
      while (ptr->next != actPose && ptr!= NULL){
         ptr = ptr->next;
      }
      if (ptr != NULL){
         ptr->next = actPose->next;
         vaSgFree(actPose);
         actData->numCurPoses--;
      }
   }
 }

static void poseActorRuntimeFunc(vaActor *act){
   int slotIndex           = poseGetExtIdentifier();
   vaActclass *ac      = vaActorGetActclass(act);
   poseActclassData *acData = (poseActclassData *)vaActclassGetExtData(ac,  slotIndex);
   poseActorData   *actData  = (poseActorData *)   vaActorGetExtData(   act, slotIndex);
   float curTime       = vaGetCurTime();
   vaxActPose *actPose = actData->curPoses;
   while (actPose != NULL){
      vaxPose *pose = acData->poses[actPose->poseIndex];
      float  value  = (curTime - actPose->startTime)/actPose->transitionTime;
      if (value <= 1.0f){
         for (int i = 0; i <pose->numDofs; i++){
            vaActorSetDofValueGradual(act, pose->dofIndexes[i],   pose->dofValues[i], value);
         }
         actPose = actPose->next;
      }
      else{
          poseRemoveFromActor(act, actPose);
          actPose = actPose->next;
      }
   }
}
```

El bloque anterior contiene las funciones estáticas del fichero, encargadas básicamente en funciones de crear y liberar la memoria de las estructuras comentadas anteriormente. La función *poseCreateGData()*

rellena el puntero global asignándole un identificador adecuado. Dicho identificador es retornado por la función *poseGetExtIdentifier()*.

Las funciones *poseActclassDataCreate()*, *poseActorDataCreate()*, *poseActclassDataDelete()* y *poseActorDataDelete()* crean y liberan la memoria de las estructuras *poseActclassData* y *poseActData*.

La función *poseRemoveFromActor()* quita una pose de la lista de poses actuales de un *vaActor.* Es llamada automáticamente cuando el actor ha alcanzado los valores finales indicados en la pose.

La función *poseActorRuntimeFunc()* recorre la lista de poses de un actor actualizando de forma coherente sus grados de libertad y borrando las poses que han alcanzado sus valores finales.

```
//---- Initialization API ---------------------------------------------------//
vaExtension *vaxPoseInit( ){
   vaExtension *ext  = vaExtensionNew("pose");
   int extIdentifier = vaAddExtension(ext);
   poseCreateGData( extIdentifier );
   vaExtensionSetCreateFuncs(ext, poseActclassDataCreate, poseActorDataCreate, NULL, NULL);
   vaExtensionSetDeleteFuncs(ext, poseActclassDataDelete,  poseActorDataDelete, NULL, NULL);
   vaExtensionSetActorUpdateFunc(    ext, poseActorRuntimeFunc);
   return ext;
}
```

En la función de inicialización *vaxPoseInit()*, se crea una nueva extensión mediante la función *vaExtensiónNew(),* y a continuación se solicita a la librería de actores un identificador único mediante la función *vaExtensionGetIdentifier()*. A continuación se fijan las rutinas de creación y borrado de la información existente en los *extension slots* de la *vaActclass* y el *vaActor.* Como se puede ver, las funciones relacionadas con el *vaDof* y el *vaSkelprot* son rellenadas con valores *NULL*. Por último, mediante la función *vaExtensionSetActorUpdateFunc(),* se fija la función encargada de actualizar el estado de los actores que estén ejecutando la transición hacia alguna pose.

```
//---- Actclass API ---------------------------------------------------------//
int  vaxActclassPoseCreate(vaActclass *ac, int numDofs, int *dofIndexes,
                float *dofValues){
   vaxPose *pose      = (vaxPose*)vaSgMalloc(sizeof(vaxPose));
   pose->numDofs    = numDofs;
   pose->dofIndexes = (int   *)vaSgMalloc(numDofs*sizeof(int)  );
   pose->dofValues  = (float *)vaSgMalloc(numDofs*sizeof(float));
   memcpy(pose->dofIndexes, dofIndexes, numDofs*sizeof(int));
   memcpy(pose->dofValues,  dofValues,  numDofs*sizeof(float));

   int slotIndex           = poseGetExtIdentifier();
   poseActclassData *acData =  (poseActclassData*)vaActclassGetExtData(ac, slotIndex);
   acData->numPoses ++;
   if (acData->poses == NULL){
        acData->poses = (vaxPose **)vaSgMalloc(acData->numPoses * sizeof(vaxPose*));
        }
   else{
        acData->poses = (vaxPose **)vaSgRealloc(acData->poses,   acData->numPoses*sizeof(vaxPose*));
```

```
        }
    acData->poses[acData->numPoses -1] = pose;
    return acData->numPoses-1;
}

void vaxActclassPoseDelete(vaActclass *ac, int poseIndex){
    int slotIndex           = poseGetExtIdentifier();
    poseActclassData *acData = (poseActclassData*)vaActclassGetExtData(ac, slotIndex);
    vaxPose *pose           = acData->poses[poseIndex];
    vaSgFree(pose->dofIndexes);
    vaSgFree(pose->dofValues);
    vaSgFree(pose);
    acData->numPoses--;
    for (int i = poseIndex; i <acData->numPoses-1; i++)
        acData->poses[i] = acData->poses[i+1];
}
```

La función *vaxActclassPoseCreate()* permite añadir una nueva pose a la lista de poses de un *vaActclass*, se le pasa como parámetros el número de *dofs* a los que afecta, y a continuación un array con los índices de los *dofs* y otro de igual tamaño con sus valores finales. Retorna un índice que actúa como identificador para esa postura. La función *vaxActclassPoseDelete()* elimina una postura de la lista de posturas del *vaActclass*. Se puede observar como internamente emplean la función *poseGetExtIdentifier()* para determinar a que *extension slot* se ha de acceder para obtener la información que necesitan.

```
//---- Actor API ---------------------------------------------------------//
int vaxActorPoseApply(vaActor *act, int poseIndex, float transTime){
    int slotIndex          = poseGetExtIdentifier();
    poseActorData *actData  = (poseActorData *) vaActorGetExtData( act, slotIndex);
    actData->numCurPoses++;
    vaxActPose *actPose     = (vaxActPose*)vaSgMalloc( sizeof(vaxActPose) );

    actPose->poseIndex        = poseIndex;
    actPose->startTime        = vaGetCurTime();
    actPose->transitionTime   = transTime;
    actPose->next             = NULL;
    if (actData->curPoses == NULL)
        actData->curPoses   = actPose;
    else{
        vaxActPose *ptr = actData->curPoses;
        while (ptr->next != NULL)
            ptr = ptr->next;
        ptr->next = actPose;
    }
    return actData->numCurPoses -1;
}
```

Por último, la función *vaxActorPoseApply()* permite indicar a un *vaActor* que se quiere que adopte cierta postura, y en que tiempo se desea que adopte el estado final.

9.5.4.2 Extensión de gestión de *Keyframing*.

La extensión para soporte del *keyframing* es muy similar a la de posturas, permitiendo definir no solamente una postura sino una secuencia de posturas que serán aplicadas secuencialmente durante un periodo de tiempo. Al igual que en el caso de las poses, un actor puede estar ejecutando simultáneamente varias tablas de *keyframing*.

El fichero de cabecera de la extensión de gestión de tablas de *keyframing* se llama "*vaxKftable.h"* y su contenido es el siguiente:

Fichero vaxKftable.h

```
#ifndef __VAX_KFTABLE_H__
#define __VAX_KFTABLE_H__
#include "vaApi.h"

//---- Initialization API ------------------------------------------------//
extern vaExtension *vaxKftableInit();

//---- Actclass API ------------------------------------------------------//
extern int  vaxActclassKftableCreate(   vaActclass *ac, int numDofs,  int numKframes, int *dofIndexes,
                                        float *kfTimes, float *dofValues,  float offDistance);
extern void vaxActclassKftableDelete(vaActclass *ac, int kftIndex);

//---- Actor API ---------------------------------------------------------//
extern int  vaxActorKftableAdd(vaActor *act, int kftIndex);
extern void vaxActorKftableRemove(vaActor *act, int kftIndex);
extern void vaxActorKftableUpdateSklsroot(vaActor *act, int kftIndex);

#endif
```

La función *vaxActclassKftableCreate()* crea una nueva tabla de *keyframing* en un *vaActclass*, se le indica el número de *dofs* a los que afecta, el número de *keyframes*, es decir, el número de instantes temporales en los que se redefinirán los valores de los grados de libertad, un puntero a un array que contiene los índices de los *dofs* implicados, otro que contiene los instantes temporales en los que están definidos los *keyframes*, y por último, un puntero a un array bidimensional que contiene los valores de los *dofs* en los distintos instantes de tiempo. Como resultado esta función retorna el índice con el que se identificara a la tabla recién creada.

La función *vaxActclassKftableDelete()* elimina una tabla de *keyframes* de la lista existente en el *vaActclass*.

Las funciones *vaxActorKftableAdd()* y *vaxActorKftableRemove()* permiten añadir o quitar una tabla de *keyframing* de la lista de tablas en ejecución que tiene el *vaActor*.

Por último, la función *vaxActorKftableUpdateSklsroot()* permite que una tabla de *keyframing* actúe modificando únicamente los valores relacionados con el *Skeletons Root* del actor. Esto es de utilidad para

poder actualizar la posición de la *Sklsroot Bsphere* con objeto de realizar adecuadamente las operaciones de culling de los actores virtuales.

Fichero vaxKftable.c

```
#include "stdlib.h"
#include <math.h>
#include <string.h>

#include "vaApi.h"

//---------Kftable extension slot ----------------------------------------//
typedef struct kftableGobalDataStr{
   int extIdentifier;
}kftableGlobalData;

kftableGlobalData *kftableGdata;

typedef struct vaxKftableStr{
   int     numDofs;
   int    *dofIndexes;          // Indices de los dofs a los afecta
   int     numKfTimes;
   float  *kfTimes;
   float  *dofValues;           // Tabla de 2 dimensiones con los valores
   float   totalTime;           // Duracion total de la tabla.
   float   offDistance;         // Distancia de desactivacion de la tabla (LOD).
}vaxKftable;

typedef struct vaxActKftableStr{
   int    kftIndex;             // Indice de la kftable en la lista del actclass.
   float  startTime;            // Instante de inicio de ejecucion de la tabla.
   struct vaxActKftableStr *next;
}vaxActKftable;

typedef struct kftableActclassDataStr{
   int        numKftables;
   vaxKftable   **kftables;   // Lista de kfTables.
}kftableActclassData;

typedef struct kftableActorDataStr{
   int        numCurKftables;
   vaxActKftable *curKftables;
}kftableActorData;

static void kftableCreateGData(int extIdentifier){
   kftableGdata   = (kftableGlobalData*)vaSgMalloc(sizeof(kftableGlobalData));
   kftableGdata->extIdentifier = extIdentifier;
}

static int kftableGetExtIdentifier(){
   return kftableGdata->extIdentifier;
}

static void *kftableActclassDataCreate(void *data){
   vaActclass *ac = (vaActclass *)data;
   kftableActclassData *acData;
   acData = (kftableActclassData *)vaSgMalloc(sizeof(kftableActclassData));
   acData->numKftables        = 0;
   acData->kftables           = NULL;
```

```
    return acData;
}

static void *kftableActorDataCreate(void *data){
    vaActor *act = (vaActor *)data;
    kftableActorData *actData;
    actData  = (kftableActorData*)vaSgMalloc(sizeof(kftableActorData));
    actData->numCurKftables = 0;
    actData->curKftables        = NULL;
    return actData;
}

static void kftableActclassDataDelete(void *data){
    vaActclass *ac    = (vaActclass *)data;
    int slotIndex       = kftableGetExtIdentifier();
    kftableActclassData *acData;
    acData = (kftableActclassData *)vaActclassGetExtData(ac, slotIndex);
    for (int i = 0; i < acData->numKftables; i++)
        vaSgFree(acData->kftables[i]);
    vaSgFree(acData->kftables);
}

static void kftableActorDataDelete(void *data){
    vaActor *act       = (vaActor *)data;
    int slotIndex      = kftableGetExtIdentifier();
    kftableActorData *actData;
    actData = (kftableActorData *)vaActorGetExtData(act, slotIndex);
    vaxActKftable *ptr       = actData->curKftables;
    while (ptr != NULL){
        vaxActKftable *auxPtr = ptr;
       ptr = ptr->next;
        vaSgFree(auxPtr);
        }
    vaSgFree(actData);
}

static void kftableGetFirstKfAndTime( vaxKftable *kftable, float localTime,  float *normVal, int *firstKf){
    int numCycles = (int)floor(localTime/kftable->totalTime);
    float  value  = localTime -numCycles*kftable->totalTime;

    *firstKf = 0;
    for (int i = 1; i < kftable->numKfTimes; i++){
       if (kftable->kfTimes[i] > value){
           *firstKf = i-1;
           break;
       }
    }
    *normVal = (value - kftable->kfTimes[*firstKf])/(kftable->kfTimes[*firstKf+1] -kftable->kfTimes[*firstKf]);
}

static void kftableActorRuntimeFunc(vaActor *act){
    int slotIndex       = kftableGetExtIdentifier();
    vaActclass *ac    = vaActorGetActclass(act);
    kftableActclassData *acData;
    acData = (kftableActclassData *)vaActclassGetExtData(ac,  slotIndex);
    kftableActorData   *actData;
    actData = (kftableActorData *)   vaActorGetExtData(   act, slotIndex);

    float curTime       = vaGetCurTime();

    vaxActKftable *actKftable = actData->curKftables;
```

```
    while (actKftable != NULL){
        vaxKftable *kftable = acData->kftables[actKftable->kftIndex];
        if (vaActorGetEyeDistance(act) > kftable->offDistance)
              break;
        float localTime = curTime - actKftable->startTime;

        float cycleTime    = kftable->totalTime;
        int   numCycles    = (int)floor(localTime/cycleTime);

        if (numCycles >1){
           actKftable->startTime += numCycles*cycleTime;
           localTime = curTime - actKftable->startTime;
        }

        float t;
        int firstKf;
        kftableGetFirstKfAndTime(kftable, localTime, &t, &firstKf);

        float prevVal, nextVal;
        for (int i = 0; i <kftable->numDofs; i++){
            int dofIndex = kftable->dofIndexes[i];
            if (dofIndex >= 0){
              prevVal = kftable->dofValues[    firstKf*kftable->numDofs +i];
              nextVal = kftable->dofValues[(firstKf+1)*kftable->numDofs +i];
              vaActorSetDofValue(act, dofIndex, prevVal +t*(nextVal -prevVal));
            }
        }
        actKftable = actKftable->next;
    }
}

//---- Initialization API ---------------------------------------------------//
vaExtension *vaxKftableInit( ){
    vaExtension *ext  = vaExtensionNew("Kftable");
    int extIdentifier     = vaAddExtension(ext);
    kftableCreateGData( extIdentifier );
    vaExtensionSetCreateFuncs(ext, kftableActclassDataCreate, kftableActorDataCreate, NULL, NULL);
    vaExtensionSetDeleteFuncs(ext, kftableActclassDataDelete,  kftableActorDataDelete, NULL, NULL);
    vaExtensionSetActorUpdateFunc(    ext, kftableActorRuntimeFunc);
    return ext;
}

//---- Actclass API ---------------------------------------------------------//
int  vaxActclassKftableCreate(vaActclass *ac, int numDofs, int numKframes,
                              int *dofIndexes, float *kfTimes, float *dofValues,   float offDistance){
    vaxKftable *kftable          = (vaxKftable*)vaSgMalloc(sizeof(vaxKftable));
    kftable->numDofs             = numDofs;
    kftable->numKfTimes          = numKframes;
    kftable->dofIndexes          = (int    *)vaSgMalloc(numDofs*sizeof(int)     );
    kftable->kfTimes             = (float *)vaSgMalloc(numKframes*sizeof(float));

    kftable->dofValues           = (float *)vaSgMalloc(numDofs*numKframes*sizeof(float));
    kftable->totalTime           =  kfTimes[numKframes -1];
    kftable->offDistance         = offDistance;

    memcpy(kftable->dofIndexes, dofIndexes, numDofs*sizeof(int));
    memcpy(kftable->kfTimes,    kfTimes,    numDofs*sizeof(float));
    memcpy(kftable->dofValues,  dofValues,  numDofs*numKframes*sizeof(float));

    int slotIndex              = kftableGetExtIdentifier();
    kftableActclassData *acData;
```

```
    acData = (kftableActclassData*)vaActclassGetExtData(ac, slotIndex);
    acData->numKftables ++;
    if (acData->kftables == NULL){
        acData->kftables = (vaxKftable **)vaSgMalloc(acData->numKftables* sizeof(vaxKftable*));
        }
    else{
        acData->kftables = (vaxKftable **)vaSgRealloc(acData->kftables, acData->numKftables*sizeof(vaxKftable*));
        }
    acData->kftables[acData->numKftables -1] = kftable;
    return acData->numKftables-1;
}

void vaxActclassKftableDelete(vaActclass *ac, int kftIndex){
    int slotIndex               = kftableGetExtIdentifier();
    kftableActclassData *acData = (kftableActclassData*)vaActclassGetExtData(ac, slotIndex);
    vaxKftable *kftable         = acData->kftables[kftIndex];
    vaSgFree(kftable->dofIndexes);
    vaSgFree(kftable->kfTimes);
    vaSgFree(kftable->dofValues);
    vaSgFree(kftable);
    acData->numKftables--;
    for (int i = kftIndex; i <acData->numKftables-1; i++)
        acData->kftables[i] = acData->kftables[i+1];
}

//---- Actor API ----------------------------------------------------------//
int vaxActorKftableAdd(vaActor *act, int kftIndex){
    int slotIndex            = kftableGetExtIdentifier();
    kftableActorData *actData  = (kftableActorData *) vaActorGetExtData( act, slotIndex);
    actData->numCurKftables++;
    vaxActKftable *actKftable;
    actKftable  = (vaxActKftable*)vaSgMalloc( sizeof(vaxActKftable) );

    actKftable->kftIndex        = kftIndex;
    actKftable->startTime       = vaGetCurTime();
    actKftable->next            = NULL;
    if (actData->curKftables == NULL)
       actData->curKftables   = actKftable;
    else{
       vaxActKftable *ptr = actData->curKftables;
       while (ptr->next != NULL)
          ptr = ptr->next;
       ptr->next = actKftable;
    }
    return actData->numCurKftables -1;
}

void vaxActorKftableRemove(vaActor *act, int kftIndex){
    int slotIndex = kftableGetExtIdentifier();
    vaActclass *ac          = vaActorGetActclass(act);
    kftableActorData *actData = (kftableActorData *) vaActorGetExtData( act, slotIndex);
    kftableActclassData *acData;
    acData = (kftableActclassData *)vaActclassGetExtData(ac,  slotIndex);

    vaxActKftable *actKftable = actData->curKftables;
    while (actKftable != NULL && actKftable->kftIndex != kftIndex){
       actKftable = actKftable->next;
    }
    if (actKftable == NULL)
      return;
```

```
    if (actData->curKftables == actKftable){
        actData->curKftables =  actKftable->next;
        vaSgFree( actKftable );
        actData->numCurKftables--;
    }
    else{
        vaxActKftable *ptr = actData->curKftables;
        while (ptr->next != actKftable && ptr!= NULL){
            ptr = ptr->next;
        }
        if (ptr != NULL){
            ptr->next = actKftable->next;
            vaSgFree(actKftable);
            actData->numCurKftables--;
        }
    }
}

void vaxActorKftableUpdateSklsroot(vaActor *act, int kftIndex){
    int slotIndex       = kftableGetExtIdentifier();
    vaActclass *ac    = vaActorGetActclass(act);
    kftableActclassData *acData = (kftableActclassData *)vaActclassGetExtData(ac,  slotIndex);
    kftableActorData   *actData;
    actData = (kftableActorData *)   vaActorGetExtData(   act, slotIndex);

    vaxActKftable * actKftable     = NULL;
    vaxActKftable *actKftablePtr  = actData->curKftables;
    while (actKftablePtr != NULL && actKftable == NULL){
      if (actKftablePtr->kftIndex == kftIndex)
           actKftable = actKftablePtr;
      actKftablePtr = actKftablePtr->next;
    }
    if (actKftable == NULL)
      return;

    vaxKftable *kftable  = acData->kftables[kftIndex];
    float curTime          = vaGetCurTime();
    float localTime        = curTime - actKftable->startTime;

    float t, curVal, prevVal, nextVal;
    int firstKf;

    kftableGetFirstKfAndTime(kftable, localTime, &t, &firstKf);

    for (int i = 0; i <kftable->numDofs; i++){
        int dofIndex = kftable->dofIndexes[i];
        if (dofIndex < 0){ //-Afecta a sklsRoot
            prevVal = kftable->dofValues[    firstKf*kftable->numDofs +i];
            nextVal = kftable->dofValues[(firstKf+1)*kftable->numDofs +i];
            curVal  = prevVal +t*(nextVal -prevVal);
            vaActorSetDofValue(act, dofIndex, curVal);
        }
    }
}
```

Como se puede observar, el código del fichero *"vaxKftable.c"* esta estructurado de un modo similar al *"vaxPose.c"*. Como se puede observar, *vaActclass* emplea la estructura *kftableActclassData* para almacenar una lista de estructuras *vaxActKftable,* cada una de las cuales almacena una tabla de

keyframing que puede ser aplicada a los actores de esa clase. Las tablas que un *vaActor* está ejecutando en un momento determinado son almacenadas dentro de la estructura *kftableActorData*, que como se puede observar, contiene una lista enlazada de estructura de tipo *vaxActKftable*, cada una de ellas almacena información sobre el uso particular que un actor hace de una tabla de *keyframing* (básicamente el índice de la tabla y el instante en el que comenzó a ejecutarse).

No se va a entrar a explicar en más detalle el contenido de este fichero, se ha procurado escribir el código de un modo auto-explicativo, e incorporar algún comentario en las zonas que presentan alguna característica especial. Respecto a este módulo de gestión de *keyframing*, presenta unas capacidades mínimas (puesto que tan sólo pretende actuar como ejemplo), pudiendo ser ampliado en el futuro de diferentes modos, permitiendo por ejemplo la interpolación entre varias tablas, incorporando controles sobre la velocidad de ejecución, etc. Es oportuno llamar la atención sobre la facilidad con la que algo relativamente complejo como puede ser la implementación de un módulo de *keyframing*, ha sido integrado sobre a las estructuras básicas propuestas en este trabajo, siendo éste un buen ejemplo para mostrar la conveniencia de mostrar una división entre *vaActor* y *vaActclass,* y también, para mostrar la forma en el mecanismo de extensión mediante la clase *vaExtension* y los *extension slots,* permite ampliar fácilmente las capacidades básicas de la librería de gestión de actores.

9.5.4.3 Extensión de gestión del Parpadeo.

El mecanismo de control del parpadeo automatiza el control de los párpados del actor. Esta extensión emplea internamente a la extensión de posturas *vaxPose* explicada anteriormente.

El fichero de cabecera de la extensión de gestión del parpadeo se llama "*vaxBlink.h"* y su contenido es el siguiente:

```
#ifndef __VAX_BLINK_H__
#define __VAX_BLINK_H__
#include "vaApi.h"

//---- Initialization API ----------------------------------------------------//
extern vaExtension *vaxBlinkInit();

//---- Actclass API ----------------------------------------------------------//
extern void vaxActclassBlinkSetPoseIndexes( vaActclass *ac, int openPoseIndex, int closePoseIndex);
extern void vaxActclassBlinkSetLodDistance( vaActclass *ac, float offDistance);

//---- Actor API -------------------------------------------------------------//
extern void vaxActorBlinkSetSpeeds(vaActor *act, float openTimeMin,  float openTimeMax,
                                                 float waitTimeMin,    float waitTimeMax);
extern void vaxActorBlinkSetAttention(vaActor *act, float attention);
#endif
```

Como se puede observar, las funciones de ampliación del API de actores son muy sencillas. Respecto a la *vaActclass* se define la función ***vaxActclassBlinkSetPoseIndexes()***, a la que se le indica los índices de las posturas que colocan al actor con los ojos cerrados y con los ojos abiertos. Así pues, previa a la llamada a esta función es necesario haber añadido al *vaActclass* estas dos posturas y obtenidos sus índices. La función ***vaxActclassBlinkSetLodDistance()*** sirve para fijar la distancia de desactivación del mecanismo de gestión de parpadeo, esta función permite controlar el nivel de detalle de comportamiento, puesto que permite reducir el coste computacional del procesado de aquellos actores que se encuentran tan lejos de la cámara como para que su parpadeo resulte inapreciable. Respecto al *vaActor* tan sólo se definen las funciones ***vaxActorBlinkSetSpeeds()***, que permite definir los tiempos mínimos y máximos de apertura o cierre de los párpados, y también el tiempo mínimo y máximo entre parpadeos; el hecho de que estos valores estén entre unos márgenes permite que el parpadeo presente un comportamiento aleatorio más realista. La función ***vaxActorBlinkSetAttention()*** permite modular el parpadeo, haciendo que sea más rápido o más lento dependiendo del estado de atención del actor. Estas dos funciones permiten que cada actor virtual, pese a pertenecer a una misma clase de actor, pueda mostrar un comportamiento personalizado.

A continuación se muestra el contenido del fichero que contiene el código que implementa esta extensión:
Fichero vaxBlink.c

```
#include "stdlib.h"
#include <stdio.h>
#include "vaApi.h"
#include "vaxPose.h"
```

```
//------- Blinking extension slot   ----------------------------------------//
#define BLINKSTATE_WAITING   0
#define BLINKSTATE_CLOSING   1
#define BLINKSTATE_OPENING   2

typedef struct blinkGobalDataStr{
   int extIdentifier;
}blinkGlobalData;

blinkGlobalData *blinkGdata;

typedef struct blinkActclassDataStr{
   int    eyelidsOpenIndex;
   int    eyelidsCloseIndex;
   float offDistance;
}blinkActclassData;

typedef struct blinkActorDataStr{
   float totalTimeMax;
   float totalTimeMin;
   float openTimeMin;
   float openTimeMax;;
   float waitTimeMin;
   float waitTimeMax;
   float openTime;  //Tiempo ocupado en la apertura de los parpados.
   float waitTime;    //Tiempo entre parpadeos.
   float totalTime;
   float startTime;
   int   state;
   float attention;    //0 < x < 1
}blinkActorData;

static void blinkCreateGData(int extIdentifier){
   blinkGdata = (blinkGlobalData*)vaSgMalloc(sizeof(blinkGlobalData));
   blinkGdata->extIdentifier = extIdentifier;
}

static int blinkGetExtIdentifier(){
   return blinkGdata->extIdentifier;
}

static void *blinkActclassDataCreate(void *data){
   blinkActclassData *acData;
   acData = (blinkActclassData *)vaSgMalloc(sizeof(blinkActclassData));
   acData->eyelidsOpenIndex = -1;
   acData->eyelidsCloseIndex = -1;
   return  acData;
}

static void *blinkActorDataCreate(void *data){
   vaActor *act = (vaActor *)data;
   blinkActorData *actData;
   actData  = (blinkActorData *)vaSgMalloc(sizeof(blinkActorData));
   actData->openTimeMin     = 0.2f;
   actData->openTimeMax     = 0.5f;
   actData->waitTimeMin     = 0.2f;
   actData->waitTimeMax     = 2.0f;
   actData->state           = BLINKSTATE_WAITING;
   actData->attention       = 0.5f;
   return  actData;
}
```

```
static void blinkActclassDataDelete(void *data ){
   vaActclass *ac    = (vaActclass*)data;
   int slotIndex     = blinkGetExtIdentifier();
   blinkActclassData *acData;
   acData            = (blinkActclassData*)vaActclassGetExtData(ac, slotIndex);
   vaSgFree(acData);
}

static void blinkActorDataDelete(void *data){
   vaActor *act      = (vaActor*)data;
   int slotIndex     = blinkGetExtIdentifier();
   blinkActorData *actData ;
   actData           = (blinkActorData*)vaActorGetExtData(act, slotIndex);
   vaSgFree(actData);
}

static float randomValInMargin(float min, float max){
    float normRand = (float)rand()/RAND_MAX;
    return min + (max-min)*normRand;
}

static void blinkActorRuntimeFunc(vaActor *act){
   vaActclass *ac    = vaActorGetActclass(act);
   int slotIndex     = blinkGetExtIdentifier();
   blinkActclassData *acData= (blinkActclassData *)vaActclassGetExtData(ac,  slotIndex);
   blinkActorData    *actData = (blinkActorData *)   vaActorGetExtData(   act, slotIndex);

   float distance    = vaActorGetEyeDistance(act);

   if (distance > acData->offDistance)//deativation distance.
       return;

   float curTime     = vaGetCurTime();

   float relTime     = curTime -actData->startTime;
   switch (actData->state) {
      case BLINKSTATE_WAITING:
          if (relTime >= actData->totalTime){
            float openTimeRange = actData->openTimeMax -actData->openTimeMin;
            float openTimeMax   = actData->openTimeMin +  (1.0f-actData->attention)*openTimeRange;
            actData->openTime   = randomValInMargin(actData->openTimeMin,  openTimeMax);

            float waitTimeRange = actData->waitTimeMax -actData->waitTimeMin;
            float waitTimeMin   = actData->waitTimeMin + actData->attention*waitTimeRange;
            actData->waitTime   = randomValInMargin(waitTimeMin,  actData->waitTimeMax);

            actData->totalTime = 2*actData->openTime + actData->waitTime;
            actData->startTime = vaGetCurTime();
            vaxActorPoseApply(act, acData->eyelidsCloseIndex,   actData->openTime);
            actData->state = BLINKSTATE_CLOSING;
          }
          break;
      case BLINKSTATE_CLOSING:
          if (relTime >= actData->openTime){
            vaxActorPoseApply(act, acData->eyelidsOpenIndex,  actData->openTime);
            actData->state = BLINKSTATE_OPENING;
          }
          break;
      case BLINKSTATE_OPENING:
          if (relTime >= 2*actData->openTime){
```

```
            actData->state = BLINKSTATE_WAITING;
         }
         break;
   }
}

//---- Initialization API ----------------------------------------------------//
vaExtension *vaxBlinkInit(){
   vaExtension *ext  = vaExtensionNew("blinking");
   int extIdentifier = vaAddExtension(ext);
   blinkCreateGData( extIdentifier);
   vaExtensionSetCreateFuncs(  ext, blinkActclassDataCreate, blinkActorDataCreate, NULL, NULL);
   vaExtensionSetDeleteFuncs(  ext, blinkActclassDataDelete,  blinkActorDataDelete, NULL, NULL);
   vaExtensionSetActorUpdateFunc( ext, blinkActorRuntimeFunc);
   return ext;
}

//---- Actclass API ----------------------------------------------------------//
void vaxActclassBlinkSetPoseIndexes( vaActclass *ac,  int openPoseIndex, int closePoseIndex){
   int slotIndex = blinkGetExtIdentifier();
   blinkActclassData * acData;
   acData  = (blinkActclassData *)vaActclassGetExtData(ac,slotIndex );
   acData->eyelidsOpenIndex   = openPoseIndex;
   acData->eyelidsCloseIndex  = closePoseIndex;
}

void vaxActclassBlinkSetLodDistance(vaActclass *ac, float offDistance){
   int slotIndex = blinkGetExtIdentifier();
   blinkActclassData * acData;
   acData  = (blinkActclassData *)vaActclassGetExtData(ac, slotIndex);
   acData->offDistance = offDistance;
}

//---- Actor API -------------------------------------------------------------//
void vaxActorBlinkSetSpeeds(vaActor *act, float openTimeMin, float openTimeMax,
                  float waitTimeMin, float waitTimeMax){
   int slotIndex = blinkGetExtIdentifier();
   blinkActorData * actData = (blinkActorData *)vaActorGetExtData(act, slotIndex);
   actData->openTimeMin = openTimeMin;
   actData->openTimeMax = openTimeMax;
   actData->waitTimeMin = waitTimeMin;
   actData->waitTimeMax = waitTimeMax;
   actData->startTime   = vaGetCurTime();
}

void vaxActorBlinkSetAttention(vaActor *act, float attention){
   int slotIndex = blinkGetExtIdentifier();
   blinkActorData * actData  = (blinkActorData *)vaActorGetExtData(act, slotIndex);
   actData->attention = attention;
}
```

9.5.4.4 Extensión de gestión de la Mirada.

La extensión de gestión de la mirada se encarga de orientar la mirada del actor virtual hacia un determinado punto, actuando para ello sobre los grados de libertad del cuello y los ojos. En la definición de esta extensión, además de los ficheros principales *"vaxLook.h"* y *"vaxLook.c"*, también se emplea un fichero auxiliar que contiene algunas funciones relacionadas resolución de cálculos de cinemática inversa, el fichero *"ikLookFuncs.h"* contiene los prototipos de estas funciones.

El contenido del fichero *"vaxLook.h"* es el siguiente:

```
#ifndef __VAX_LOOK_H__
#define __VAX_LOOK_H__

#include "vaApi.h"

//---- Initialization API ------------------------------------------------//
vaExtension *vaxLookInit();

//---- Actclass API ------------------------------------------------------//
void vaxActclassLookSetDofIndexes(   vaActclass *ac,    int headAzimIndex, int headElevIndex,int headTwistIndex,
                                     int eyeLeftAzimIndex,  int eyeLeftElevIndex,
                                     int eyeRightAzimIndex, int eyeRightElevIndex);
void vaxActclassLookSetDofsLimits(vaActclass *ac, float headAzimMin, float headAzimMax,
                                     float headElevMin, float headElevMax,
                                     float eyesAzimMin, float eyesAzimMax,
                                     float eyesElevMin, float eyesElevMax);

void vaxActclassLookSetSklIndexes(  vaActclass *ac, int headSklIndex,
                                     int eyeLSklIndex, int eyeRSklIndex);

void vaxActclassLookSetLodDistances( vaActclass *ac,
                                     float d1, float d2, float d3, float d4);

//---- Actor API ---------------------------------------------------------//
void vaxActorLookSetSpeeds( vaActor *act, float headSpeed, float eyesSpeed);
void vaxActorLookSetFirstTarget( vaActor *act, float3 target);
void vaxActorLookSetSecondTarget(vaActor *act, float3 target);
void vaxActorLookSetActiveTarget(vaActor *act, int activeTarget);
#endif
```

Para configurar a una *vaActclass* para que pueda emplear la extensión de control de la mirada hace falta indicarle los índices de los grados de libertad que controlan los tres grados de libertad del cuello, y también los dos grados de libertad de cada uno de los ojos, esto se hace por medio de la función ***vaxActclassLookSetDofIndexes()***, además, hay que indicar cuales son los topes de azimut y elevación del cuello y un ojo, esto se hace mediante la función ***vaxActclassLookSetDofsLimits()***. También es necesario proporcionar los índices de los puntos de articulación del cuello y los ojos mediante la función ***vaxActclassLookSetSklIndexes()***. Por último, mediante la función ***vaxActclassLookSetLodDistances()*** se definen las distancias de cambio de nivel de detalle del mecanismo de mirada. El mecanismo de mirada emplea cuatro métodos alternativos para calcular la orientación del cuello y los ojos, unos son muy precisos pero lentos y otros presentan poca precisión, pero tienen un reducido coste computacional.

Las funciones que amplían el API del *vaActor* para el control de la mirada son cuatro: ***vaxActorLookSetSpeeds()*** que define la rapidez con la que los ojos y la cabeza se mueven buscando el objetivo; las funciones ***vaxActorLookSetFirstTarget()*** y ***vaxActorLookSetSecondTarget()*** que definen el punto de vista principal y secundario, y la función ***vaxActorLookSetActiveTarget()*** que permite seleccionar el punto objetivo actual (valor 1 para el principal, o 2 para el secundario).

Como se ha comentado anteriormente, el módulo de control de la mirada emplea cuatro tipo de funciones de cinemática inversa para obtener la orientación de los ojos y el cuello, los prototipos de dichas funciones están definidos en fichero *"ikLookFuncs.h"*:

```
#ifndef __IK_LOOK_FUNCS_H__
#define __IK_LOOK_FUNCS_H__
extern int ikFunction1(         float3 target,          float16 headAbsMatrix,
                                float headAzimCur, float headElevCur,
                                float eyeLAzimCur, float eyeLElevCur,
                                float eyeRAzimCur, float eyeRElevCur,
                                float16 eyeLOffsetMat, float16 eyeROffsetMat,
                                float eyesAzimMin, float eyesAzimMax,
                                float eyesElevMin, float eyesElevMax,
                                float headAzimMin, float headAzimMax,
                                float headElevMin, float headElevMax,
                                float headSpeed, float eyesSpeed, float deltaTime,
                                float *headAzim, float *headElev, float *headTwist,
                                float *eyeLAzim, float *eyeLElev,
                                float *eyeRAzim, float *eyeRElev);

extern int ikFunction2(         float3 target,          float16 headAbsMatrix,
                                float headAzimCur, float headElevCur,
                                float eyeLAzimCur, float eyeLElevCur,
                                float eyesAzimMin, float eyesAzimMax,
                                float eyesElevMin, float eyesElevMax,
                                float headAzimMin, float headAzimMax,
                                float headElevMin, float headElevMax,
                                float headSpeed, float eyesSpeed, float deltaTime,
                                float *headAzim, float *headElev, float *headTwist,
                                float *eyeLAzim, float *eyeLElev);

extern int ikFunction3(         float3 target,          float16 headAbsMatrix,
                                float eyesAzimMin, float eyesAzimMax,
                                float eyesElevMin, float eyesElevMax,
                                float headAzimMin, float headAzimMax,
                                float headElevMin, float headElevMax,
                                float *headAzim, float *headElev, float *headTwist,
                                float *eyeLAzim, float *eyeLElev);

extern int ikFunction4(         float3 target,          float16 headAbsMatrix,
                                float headAzimMin, float headAzimMax,
                                float headElevMin, float headElevMax,
                                float *headAzim, float *headElev, float *headTwist);
#endif
```

El contenido de fichero principal de control de la mirada ("*vaxLook.c*") es el siguiente:

```
#include "stdlib.h"
#include "vaApi.h"
#include "ikLookFuncs.h"
```

```
//---------Look extension slot ----------------------------------------------//
typedef struct lookGobalDataStr{
   int extIdentifier;
}lookGlobalData;
lookGlobalData *lookGdata;

typedef struct lookActclassDataStruct{
   //- dofs index ------------//
   int headElevIndex;
   int headAzimIndex;
   int headTwistIndex;
   int eyeLeftAzimIndex;
   int eyeLeftElevIndex;
   int eyeRightAzimIndex;
   int eyeRightElevIndex;

   //- head skl index ---------//
   int headSklIndex;
   int eyeLSklIndex;
   int eyeRSklIndex;

   //- dof limits ------------//
   float eyesAzimMin, eyesAzimMax;
   float headAzimMin, headAzimMax;
   float eyesElevMin, eyesElevMax;
   float headElevMin, headElevMax;

   float lodDistances[4];
}lookActclassData;

typedef struct lookActorDataStruct{
   float   eyesSpeed;          // Velocidades de giro de la cabeza y los ojos.
   float   headSpeed;
   float3  firstTarget;
   float3  secondTarget;
   int     activeTarget;       // El primero (1), el segundo (2) o ninguno(3).
   int     attention;          // Entre 0 y 1. 1=> siempre mirando a firstTarget.
   int     notVisible;
}lookActorData;

static void lookCreateGData(int extIdentifier){
   lookGdata = (lookGlobalData*)vaSgMalloc(sizeof(lookGlobalData));
   lookGdata->extIdentifier = extIdentifier;
}

static int lookGetExtIdentifier(){
   return lookGdata->extIdentifier;
}

//-------------------------------------------------------
static float randNorm(){
   return (float)rand()/RAND_MAX;
}

static void *lookActclassDataCreate(void  *data){
   lookActclassData *acData = (lookActclassData *)vaSgMalloc(sizeof(lookActclassData));
   return acData;
}


static void *lookActorDataCreate(void *data){
```

```
    vaActor *act       = (vaActor*)data;
    int slotIndex      = lookGetExtIdentifier();
    lookActorData *actData = (lookActorData*)vaSgMalloc(sizeof(lookActorData));
    vaActclass *ac     = vaActorGetActclass(act);
    lookActclassData * acData= (lookActclassData *)vaActclassGetExtData(ac, slotIndex);
    actData->eyesSpeed        =   0.0f;
    actData->headSpeed        =   0.0f;
   actData->firstTarget[0]       =10.0f;
    actData->firstTarget[1]      = 0.0f;
    actData->firstTarget[2]      = 0.0f;
    actData->secondTarget[0] =10.0f;
    actData->secondTarget[1] = 0.0f;
    actData->secondTarget[2] = 0.0f;
    actData->activeTarget       = 1;
    return actData;
}

static void lookActclassDataDelete(void *data){
    vaActclass *ac = (vaActclass*)data;
    int slotIndex  = lookGetExtIdentifier();
    lookActclassData *acData = (lookActclassData*)vaActclassGetExtData(ac, slotIndex);
    vaSgFree(acData);
}

static void lookActorDataDelete(void *data){
    vaActor *act   = (vaActor*)data;
    int slotIndex  = lookGetExtIdentifier();
    lookActorData *actData = (lookActorData*)vaActorGetExtData(act, slotIndex);
    vaSgFree(actData);
}

static void lookActorRuntimeFunc(vaActor *act){
    vaActclass *ac    = vaActorGetActclass(act);
    int slotIndex      = lookGetExtIdentifier();
    lookActclassData * acData= (lookActclassData *)vaActclassGetExtData(ac,  slotIndex);
    lookActorData * actData = (lookActorData *)vaActorGetExtData(act,slotIndex);
    float distance     = vaActorGetEyeDistance(act);

    if (distance > acData->lodDistances[3]){ //deativation distance.
       vaActorSetSklTracking(act, acData->headSklIndex, 0);
       return;
    }

    vaActorSetSklTracking(act, acData->headSklIndex, 1);

    float deltaTime = vaGetCurDeltaTime();

    float3 target;
    switch (actData->activeTarget){
       case 1: float3Copy(target, actData->firstTarget);
             break;
       case 2: float3Copy(target, actData->secondTarget);
             break;
    }

    float headAzimCur, headElevCur;
    float eyeLAzimCur, eyeLElevCur, eyeRAzimCur, eyeRElevCur;
    headAzimCur    = vaActorGetDofValue(act, acData->headAzimIndex    );
    headElevCur    = vaActorGetDofValue(act, acData->headElevIndex    );
    eyeLAzimCur    = vaActorGetDofValue(act, acData->eyeLeftAzimIndex );
    eyeLElevCur    = vaActorGetDofValue(act, acData->eyeLeftElevIndex );
```

```
eyeRAzimCur    = vaActorGetDofValue(act, acData->eyeRightAzimIndex);
eyeRElevCur    = vaActorGetDofValue(act, acData->eyeRightElevIndex);

float16 headAbsMatrix;
vaActorGetSklAbsMatrix(act, acData->headSklIndex, headAbsMatrix);

int locked;
float headAzim, headElev, headTwist;
float eyeLAzim, eyeLElev, eyeRAzim, eyeRElev;

if (distance < acData->lodDistances[0]){ //ik ambos ojos.
   //obtencion del offset de los ojos.
   float16 eyeLOffsetMat, eyeROffsetMat;
   vaActorGetSklOffsetMat(act, acData->eyeLSklIndex, eyeLOffsetMat);
   vaActorGetSklOffsetMat(act, acData->eyeRSklIndex, eyeROffsetMat);

   locked = ikFunction1(target,        headAbsMatrix,
                  headAzimCur, headElevCur,
                  eyeLAzimCur, eyeLElevCur, eyeRAzimCur, eyeRElevCur,
                  eyeLOffsetMat, eyeROffsetMat,
                  acData->eyesAzimMin, acData->eyesAzimMax,  acData->eyesElevMin, acData->eyesElevMax,
                  acData->headAzimMin, acData->headAzimMax,  acData->headElevMin, acData->headElevMax,
                  actData->eyesSpeed, actData->headSpeed, deltaTime,
                  &headAzim, &headElev, &headTwist,
                  &eyeLAzim, &eyeLElev, &eyeRAzim, &eyeRElev);

   vaActorSetDofValue(act, acData->headAzimIndex,          headAzim );
   vaActorSetDofValue(act, acData->headElevIndex,          headElev );
   vaActorSetDofValue(act, acData->headTwistIndex,         headTwist);
   vaActorSetDofValue(act, acData->eyeLeftAzimIndex,       eyeLAzim );
   vaActorSetDofValue(act, acData->eyeRightAzimIndex,      eyeRAzim );
   vaActorSetDofValue(act, acData->eyeLeftElevIndex,       eyeLElev );
   vaActorSetDofValue(act, acData->eyeRightElevIndex,      eyeRElev );
}
else if ( distance < acData->lodDistances[1]){ //ik unico ojo.
   locked = ikFunction2(target,        headAbsMatrix,
                  headAzimCur, headElevCur,
                  eyeLAzimCur, eyeLElevCur,
                  acData->eyesAzimMin, acData->eyesAzimMax,  acData->eyesElevMin, acData->eyesElevMax,
                  acData->headAzimMin, acData->headAzimMax,   acData->headElevMin, acData->headElevMax,
                  actData->eyesSpeed,  actData->headSpeed, deltaTime,
                  &headAzim, &headElev, &headTwist,
                  &eyeLAzim, &eyeLElev );

   vaActorSetDofValue(act, acData->headAzimIndex,          headAzim );
   vaActorSetDofValue(act, acData->headElevIndex,          headElev );
   vaActorSetDofValue(act, acData->headTwistIndex,         headTwist);
   vaActorSetDofValue(act, acData->eyeLeftAzimIndex,       eyeLAzim );
   vaActorSetDofValue(act, acData->eyeRightAzimIndex,      eyeLAzim );
   vaActorSetDofValue(act, acData->eyeLeftElevIndex,       eyeLElev );
   vaActorSetDofValue(act, acData->eyeRightElevIndex,      eyeLElev );
}
else if (distance < acData->lodDistances[2]){ //unico ojo, rapido.
   locked = ikFunction3(target,        headAbsMatrix,
                  acData->eyesAzimMin, acData->eyesAzimMax,  acData->eyesElevMin, acData->eyesElevMax,
                  acData->headAzimMin, acData->headAzimMax, acData->headElevMin, acData->headElevMax,
                  &headAzim, &headElev, &headTwist,
                  &eyeLAzim, &eyeLElev );

   float headSpeed = actData->headSpeed;
   float eyesSpeed = actData->eyesSpeed;
```

```
      vaActorSetDofValueSpeed( act, acData->headAzimIndex,      headAzim, headSpeed, deltaTime);
      vaActorSetDofValueSpeed( act, acData->headElevIndex,      headElev, headSpeed, deltaTime);
      vaActorSetDofValueSpeed( act, acData->headTwistIndex,     headTwist,headSpeed, deltaTime);
      vaActorSetDofValueSpeed( act, acData->eyeLeftAzimIndex,   eyeLAzim, eyesSpeed, deltaTime );
      vaActorSetDofValueSpeed( act, acData->eyeRightAzimIndex, eyeLAzim, eyesSpeed, deltaTime );
      vaActorSetDofValueSpeed( act, acData->eyeLeftElevIndex,   eyeLElev, eyesSpeed, deltaTime );
      vaActorSetDofValueSpeed( act, acData->eyeRightElevIndex,  eyeLElev, eyesSpeed, deltaTime );
   }
   else if (distance < acData->lodDistances[3]){ //solo cabeza.
      locked = ikFunction4(target,         headAbsMatrix,
                           acData->headAzimMin, acData->headAzimMax,  acData->headElevMin, acData->headElevMax,
                           &headAzim, &headElev, &headTwist);
      float headSpeed = actData->headSpeed;
      float eyesSpeed = actData->eyesSpeed;
      vaActorSetDofValueSpeed(act, acData->headAzimIndex,  headAzim, headSpeed, deltaTime);
      vaActorSetDofValueSpeed(act, acData->headElevIndex,   headElev, headSpeed, deltaTime);
      vaActorSetDofValueSpeed(act, acData->headTwistIndex, headTwist,headSpeed, deltaTime);
   }
}

//---- Initialization API ------------------------------------------------//
vaExtension *vaxLookInit(){
   vaExtension *ext  = vaExtensionNew("look");
   int extIdentifier    = vaAddExtension(ext);
   lookCreateGData( extIdentifier );
   vaExtensionSetCreateFuncs(ext, lookActclassDataCreate,  lookActorDataCreate, NULL, NULL);
   vaExtensionSetDeleteFuncs(ext, lookActclassDataDelete,   lookActorDataDelete, NULL, NULL);
   vaExtensionSetActorUpdateFunc(ext, lookActorRuntimeFunc);
   return ext;
}

//---- Actclass API ------------------------------------------------------//
void vaxActclassLookSetDofIndexes(  vaActclass *ac,
                           int headAzimIndex, int headElevIndex,int headTwistIndex,
                           int eyeLeftAzimIndex,  int eyeLeftElevIndex, int eyeRightAzimIndex, int eyeRightElevIndex){
   int slotIndex = lookGetExtIdentifier();
   lookActclassData * acData  = (lookActclassData *)vaActclassGetExtData(ac, slotIndex);
   acData->headElevIndex       = headElevIndex;
   acData->headAzimIndex       = headAzimIndex;
   acData->headTwistIndex      = headTwistIndex;
   acData->eyeLeftAzimIndex   = eyeLeftAzimIndex;
   acData->eyeLeftElevIndex    = eyeLeftElevIndex;
   acData->eyeRightAzimIndex = eyeRightAzimIndex;
   acData->eyeRightElevIndex  = eyeRightElevIndex;
}

void vaxActclassLookSetDofsLimits(vaActclass *ac,  float headAzimMin, float headAzimMax,
                                                   float headElevMin, float headElevMax,
                                                   float eyesAzimMin, float eyesAzimMax,
                                                   float eyesElevMin, float eyesElevMax){
   int slotIndex = lookGetExtIdentifier();
   lookActclassData * acData = (lookActclassData *)vaActclassGetExtData(ac, slotIndex);
   acData->headAzimMin   = headAzimMin;
   acData->headAzimMax  = headAzimMax;
   acData->headElevMin    = headElevMin;
   acData->headElevMax   = headElevMax;
   acData->eyesAzimMin   = eyesAzimMin;
   acData->eyesAzimMax  = eyesAzimMax;
   acData->eyesElevMin    = eyesElevMin;
   acData->eyesElevMax   = eyesElevMax;
}
```

```
void vaxActclassLookSetSklIndexes(  vaActclass *ac, int headSklIndex,  int eyeLSklIndex, int eyeRSklIndex){
   int slotIndex = lookGetExtIdentifier();
   lookActclassData * acData = (lookActclassData *)vaActclassGetExtData(ac, slotIndex);
   acData->headSklIndex = headSklIndex;
   acData->eyeLSklIndex = eyeLSklIndex;
   acData->eyeRSklIndex = eyeRSklIndex;
}

//-- 0  a d1 ->calculo de cinematica inversa para ambos ojos.
//   d1 a d2 ->ambos ojos con los mismos valores.
//   d2 a d3 ->cabeza y ojo muy rapida.
//   d3 a d4 ->solo la cabeza.
//     >d4   ->desactivado.
//
void vaxActclassLookSetLodDistances(vaActclass *ac,  float d1, float d2, float d3, float d4){
   int slotIndex = lookGetExtIdentifier();
   lookActclassData * acData = (lookActclassData *)vaActclassGetExtData(ac, slotIndex);
   acData->lodDistances[0] = d1;
   acData->lodDistances[1] = d2;
   acData->lodDistances[2] = d3;
   acData->lodDistances[3] = d4;
}

//---- Actor API ------------------------------------------------------------//
void vaxActorLookSetSpeeds( vaActor *act, float headSpeed, float eyesSpeed){
   int slotIndex = lookGetExtIdentifier();
   lookActorData * actData = (lookActorData *)vaActorGetExtData(act, slotIndex);
   actData->headSpeed = headSpeed;
   actData->eyesSpeed = eyesSpeed;
}

void vaxActorLookSetFirstTarget( vaActor *act, float3 target){
   int slotIndex = lookGetExtIdentifier();
   lookActorData * actData = (lookActorData *)vaActorGetExtData(act, slotIndex);
   float3Copy(actData->firstTarget, target);
}

void vaxActorLookSetSecondTarget(vaActor *act, float3 target){
   int slotIndex = lookGetExtIdentifier();
   lookActorData * actData = (lookActorData *)vaActorGetExtData(act, slotIndex);
   float3Copy(actData->secondTarget, target);
}

void vaxActorLookSetActiveTarget(vaActor *act, int activeTarget){
   int slotIndex = lookGetExtIdentifier();
   lookActorData * actData = (lookActorData *)vaActorGetExtData(act, slotIndex);
   actData->activeTarget   = activeTarget;
}
```

9.5.4.5 Extensión de gestión del Vuelo.

La extensión de gestión del vuelo controlara tres grupos de parámetros: los que definen la posición y orientación del *Reference Point*, los que controlan la posición y orientación del *Skeletons Root,* y los que actúan sobre los grados de libertad que definen la posición de las alas y la cola. El mecanismo de vuelo emplea de forma interna a la extensión de *keyframing*. En un instante anterior a la asignación de la extensión de gestión de vuelo a una *vaActclass,* se han de definir cuatro tablas de *keyframing,* cada una de ellas representando un tipo de vuelo distinto: Se ha de definir el tipo vuelo que a emplear cuando el actor está ascendiendo, otra para cuando está descendiendo, otra para cuando vuele en horizontal, y por último, una para cuando está esperando. Una vez creadas y asignadas estas tablas, el mecanismo de gestión de vuelo funciona de una forma muy sencilla, basta con indicar el punto al que se desea que se desplace el actor, y él irá describiendo la trayectoria adecuada para llegar a ese punto, seleccionando y ejecutando las tablas de *keyframing* que resulten adecuadas.

Para entender un poco mejor la forma de funcionamiento de esta extensión veamos su fichero de cabecera "*vaxFly.h*":

```
#ifndef __VAX_FLY_H__
#define __VAX_FLY_H__
#include "vaApi.h"

#define FLYSTATE_GOINGUP          0
#define FLYSTATE_GOINGDOWN        1
#define FLYSTATE_GOINGFRONT       2
#define FLYSTATE_WAITING          3

//---- Initialization API ------------------------------------------------//
extern vaExtension *vaxFlyInit();

//---- Actclass API ------------------------------------------------------//
extern void vaxActclassFlySetKftableIndexes( vaActclass *ac, int flyUpIndex,    int flyDownIndex,
                                                             int flyFrontIndex, int flyWaitIndex);
extern void vaxActclassFlySetLodDistance(    vaActclass *ac, float offDistance);
extern void vaxActclassFlySetTargetThreshold(vaActclass *ac, float threshold);

//---- Actor API ---------------------------------------------------------//
extern void  vaxActorFlySetCurPosHpr(   vaActor *act, float3 pos, float3 hpr);
extern void  vaxActorFlySetTarget(      vaActor *act, float3 target);
extern float vaxActorFlyGetTimeWaiting( vaActor *act);
extern int   vaxActorFlyGetState(       vaActor *act);
extern void  vaxActorFlyUpdateRefpoint( vaActor *act);
extern void  vaxActorFlyUpdateSklsroot( vaActor  *act);
extern void  vaxActorFlySetSpeeds(      vaActor *act,  float vHor, float vVer, float vTurn);
#endif
```

En relación con las funciones que amplían el API del *vaActclass* tan sólo existen 3 funciones, la primera de ellas ***vaxActclassFlySetKftableIndexes()*** permite indicar los índices de las tablas de *keyframing* previamente creadas y asignadas al *vaActclass* y que almacenan los distintos modos de vuelo. La función ***vaxActclassFlySetLODDistance()*** permite fijar la distancia a la cual los movimientos del *Skeletons Root*, alas y cola resulta inapreciable, y por tanto se puede desactivar la aplicación de las tablas de *keyframes*.

Por último, la función ***vaxActclassFlySetTargetThreshold()*** permite indicar a que distancia se considera que un actor ha alcanzado su objetivo.

Respecto a las funciones que extienden el API del *vaActor*, la función ***vaxActorFlySetSpeeds()*** permite indicar las velocidades a las que un actor se desplaza en horizontal y en vertical (en metros/segundo), y también la velocidad de giro (en grados /segundo). La función ***vaxActorFlySetCurPosHpr()*** permite ubicar de forma instantanea al actor con una posición y orientación determinada. La función ***vaxActorFlyGetCurPosHpr()*** retorna la posición y orientación actual del *Reference Point* de actor virtual. La función ***vaxActorFlySetTarget()*** permite indicar el punto al que se quiere que se dirija el actor virtual. ***vaxActorFlyGetState()*** retorna el estado de vuelo en el que se encuentra el actor, el valor retornado puede ser *FLYSTATE_GOINGUP*, *FLYSTATE_GOINGDOWN*, *FLYSTATE_GOINGFRONT* o *FLYSTATE_WAITING*. La función ***vaxActorFlyGetTimeWaiting()*** retorna los segundos transcurridos desde que el actor ha alcanzado su valor destino, o -1.0 si aún no ha llegado. La función ***vaxActorFlyUpdateRefpoint()*** actualiza los valores del *Reference Point* del actor, y la función ***vaxActorFlyUpdateSklsroot()*** actualiza los valores del *Skeletons Root* del actor. Estas dos funciones aparecen separadas para que el método de gestión de culling de actores las llame solamente si es necesario.

La implementación de esas funciones y todas las necesarias para la gestión del mecanismo de vuelo, se encuentran en el fichero "*vaxFly.c*". Su contenido es el siguiente:

```
#include "stdlib.h"
#include "math.h"

#include "vaApi.h"
#include  "vaxKftable.h"
#include  "vaxFly.h"

typedef struct flyGobalDataStr{
   int extIdentifier;
}flyGlobalData;

flyGlobalData *flyGdata;

typedef struct flyActclassDataStr{
   float offDistance;    // Distancia de desactivacion del mecanismo  de vuelo.
   int   flyUpIndex;     // Indices de las tablas de keyframing empleadas
   int   flyDownIndex;
   int   flyFrontIndex;
   int   flyWaitIndex;
   float targetThreshold; // Distancia a la cual se considera que se ha  alcanzado el objetivo.
}flyActclassData;

typedef struct flyActorDataStr{
   float  vHor;                  // Velocidad de movimiento horizontal.
   float  vVer;                  // Velocidad de movimiento en vertical.
   float  vTurn;                 // Velocidad de giro.
   float  timeWaitingStart;/     / Instante de alcance del objetivo.
   float3 targetPos;             // Posicion final a la que tiene que llegar.
   int    flyCurIndex;           // Tabla de kf en ejecucion.
   int    state;                 // Estado actual de vuelo.
}flyActorData;
```

```
static void flyCreateGData(int extIdentifier){
   flyGdata = (flyGlobalData*)vaSgMalloc(sizeof(flyGlobalData));
   flyGdata->extIdentifier = extIdentifier;
}

static int flyGetExtIdentifier(){
   return flyGdata->extIdentifier;
}

static void *flyActclassDataCreate(void *data){
   flyActclassData *acData;
   acData  = (flyActclassData *)vaSgMalloc(sizeof(flyActclassData));
   return acData;
}

static void *flyActorDataCreate(void *data){
   vaActor *act       = (vaActor *)data;
   vaActclass *ac     = vaActorGetActclass(act);
   int slotIndex      = flyGetExtIdentifier();
   flyActclassData *acData;
   acData  = (flyActclassData*)vaActclassGetExtData(ac, slotIndex);
   flyActorData    *actData = (flyActorData *)vaSgMalloc(sizeof(flyActorData));

   actData->vHor    = 0.0f;
   actData->vVer    = 0.0f;
   actData->vTurn   = 0.0f;

   actData->targetPos[0]= actData->targetPos[1]= actData->targetPos[2] = 0.0f;

   actData->timeWaitingStart = -1.0f;
   actData->flyCurIndex        = acData->flyWaitIndex;
   actData->state              = FLYSTATE_WAITING;
   vaxActorKftableAdd(   act, actData->flyCurIndex);
   return actData;
}

static void flyActclassDataDelete(void *data ){
   vaActclass *ac    = (vaActclass*)data;
   int slotIndex     = flyGetExtIdentifier();
   flyActclassData *acData;
   acData            = (flyActclassData*)vaActclassGetExtData(ac, slotIndex);
   vaSgFree(acData);
}

static void flyActorDataDelete(void *data){
   vaActor *act       = (vaActor*)data;
   int slotIndex      = flyGetExtIdentifier();
   flyActorData *actData = (flyActorData*)vaActorGetExtData(act, slotIndex);
   vaSgFree(actData);
}

static void flyActorRuntimeFunc(vaActor *act){

}

//---- Initialization API ------------------------------------------------//
vaExtension *vaxFlyInit(){
   vaExtension *ext = vaExtensionNew("fly");
   int extIdentifier    = vaAddExtension(ext);
   flyCreateGData( extIdentifier );
   vaExtensionSetCreateFuncs(  ext, flyActclassDataCreate,  flyActorDataCreate, NULL, NULL);
```

```
    vaExtensionSetDeleteFuncs(  ext, flyActclassDataDelete,   flyActorDataDelete, NULL, NULL);
    vaExtensionSetActorUpdateFunc( ext, flyActorRuntimeFunc);
    return ext;
}

//---- Actclass API ------------------------------------------------------//
void vaxActclassFlySetKftableIndexes( vaActclass *ac,  int flyUpIndex,    int flyDownIndex,
                                      int flyFrontIndex, int flyWaitIndex){
    int slotIndex = flyGetExtIdentifier();
    flyActclassData * acData ;
    acData = (flyActclassData *)vaActclassGetExtData(ac,slotIndex );
    acData->flyUpIndex    = flyUpIndex;
    acData->flyDownIndex  = flyDownIndex;
    acData->flyFrontIndex = flyFrontIndex;
    acData->flyWaitIndex  = flyWaitIndex;
}

void vaxActclassFlySetLodDistance(vaActclass *ac, float offDistance){
    int slotIndex = flyGetExtIdentifier();
    flyActclassData * acData;
    acData = (flyActclassData *)vaActclassGetExtData(ac, slotIndex);
    acData->offDistance = offDistance;
}

void vaxActclassFlySetTargetThreshold(vaActclass *ac, float threshold){
    int slotIndex = flyGetExtIdentifier();
    flyActclassData * acData;
    acData = (flyActclassData *)vaActclassGetExtData(ac, slotIndex);
    acData->targetThreshold = threshold;
}

//---- Actor API ---------------------------------------------------------//
void vaxActorFlySetCurPosHpr(vaActor *act, float3 pos, float3 hpr){
    vaActorSetRefpointPos(act, pos);
    vaActorSetRefpointHpr(act, hpr);
}

void vaxActorFlySetTarget(vaActor *act, float3 target){
    int slotIndex                = flyGetExtIdentifier();
    flyActorData * actData       = (flyActorData *)vaActorGetExtData(act, slotIndex);
    float3Copy(actData->targetPos, target);
    actData->timeWaitingStart = -1.0f;
}

float vaxActorFlyGetTimeWaiting(vaActor *act){
    int slotIndex                = flyGetExtIdentifier();
    flyActorData * actData       = (flyActorData *)vaActorGetExtData(act, slotIndex);
    if (actData->timeWaitingStart < 0.0f)
        return -1.0f;
    return vaGetCurTime() - actData->timeWaitingStart;
}

int vaxActorFlyGetState(vaActor *act){
    int slotIndex                = flyGetExtIdentifier();
    flyActorData * actData       = (flyActorData *)vaActorGetExtData(act, slotIndex);
    return actData->state;
}

void vaxActorFlyUpdateRefpoint(vaActor *act){
    int slotIndex                = flyGetExtIdentifier();
    flyActorData * actData       = (flyActorData *)vaActorGetExtData(act, slotIndex);
```

```
   vaActclass *ac                = vaActorGetActclass(act);
   flyActclassData *acData    = (flyActclassData *)vaActclassGetExtData(ac, slotIndex );
   float3 actCurPos, actCurHpr;
   vaActorGetRefpointPos(act, actCurPos);
   vaActorGetRefpointHpr(act, actCurHpr);

   float xDist          = actData->targetPos[0] - actCurPos[0];
   float yDist          = actData->targetPos[1] - actCurPos[1];
   float zDist          = actData->targetPos[2] - actCurPos[2];
   float horDist        = sqrtf(xDist*xDist +yDist*yDist);

   float deltaTime   = vaGetCurDeltaTime();
   float azimCur     = actCurHpr[0];
   float azimEnd     = atan2f(yDist, xDist)*180.0f/3.1415;
   float azimNew     = angleLinearGoto(azimEnd, azimCur,  actData->vTurn, deltaTime);
   actCurHpr[0] = azimNew;

   float veloX   = cosf(azimNew*3.1415/180);
   float veloY   = sinf(azimNew*3.1415/180);

   if (horDist > acData->targetThreshold){
        actCurPos[0] += veloX*actData->vHor*deltaTime;
        actCurPos[1] += veloY*actData->vHor*deltaTime;
   }

   if (zDist > 0){
        actCurPos[2] += actData->vVer * deltaTime;
        if (actCurPos[2] > actData->targetPos[2])
           actCurPos[2] = actData->targetPos[2];
   }
   else{
        actCurPos[2] -= actData->vVer * deltaTime;
        if (actCurPos[2] < actData->targetPos[2])
           actCurPos[2] = actData->targetPos[2];
   }

  if ( fabs(zDist) > acData->targetThreshold){
        if (zDist >0)   actData->state = FLYSTATE_GOINGUP;
       else           actData->state = FLYSTATE_GOINGDOWN;
   }
   else if (horDist < acData->targetThreshold){
        if (actData->state !=  FLYSTATE_WAITING)
           actData->timeWaitingStart = vaGetCurTime();
      actData->state = FLYSTATE_WAITING;
   }
   else
      actData->state = FLYSTATE_GOINGFRONT;


   if (actData->state != FLYSTATE_WAITING){
      vaActorSetRefpointPos(act, actCurPos);
      vaActorSetRefpointHpr(act, actCurHpr);
   }
}

void vaxActorFlyUpdateSklsroot(vaActor  *act){
   int slotIndex                 = flyGetExtIdentifier();
   vaActclass *ac                = vaActorGetActclass(act);
   flyActorData * actData        = (flyActorData *)vaActorGetExtData(act, slotIndex);
   flyActclassData * acData;
   acData  = (flyActclassData *)vaActclassGetExtData(ac,slotIndex );
```

```
   int flyIndex;
   switch (actData->state){
      case FLYSTATE_GOINGUP:
           flyIndex  = acData->flyUpIndex;
           break;
       case FLYSTATE_GOINGDOWN:
            flyIndex = acData->flyDownIndex;
           break;
      case FLYSTATE_GOINGFRONT:
         flyIndex  = acData->flyFrontIndex;
         break;
      case FLYSTATE_WAITING:
          flyIndex = acData->flyWaitIndex;
         break;
   }
   if (flyIndex != actData->flyCurIndex){
      vaxActorKftableRemove(act, actData->flyCurIndex);
      actData->flyCurIndex = flyIndex;
      vaxActorKftableAdd( act, actData->flyCurIndex);
   }
   vaxActorKftableUpdateSklsroot(act, actData->flyCurIndex);
   }

void vaxActorFlySetSpeeds(vaActor *act, float vHor, float vVer, float vTurn){
   int slotIndex              = flyGetExtIdentifier();
   flyActorData *actData      = (flyActorData*)vaActorGetExtData(act, slotIndex);
   actData->vHor              = vHor;
   actData->vVer              = vVer;
   actData->vTurn             = vTurn;
   }
```

Se puede observar que la función *flyActorRuntimeFunc()* pasada a la función *vaExtensionSetActorUpdateFunc()* no realiza ninguna operación, eso es debido a que los valores del *Reference Point* de los actores ha de ser actualizado de forma externa mediante la llamada a la función *vaxActorFlyUpdateRefpoint()*, y los valores del *Skeletons Root* y los *dofs* implicados en el vuelo son actualizados directamente desde la extensión de *Keyframing*. También se puede observar como es la propia función *vaxActorFlyUpdateRefpoint()* la que se encarga de guiar automáticamente al actor hacia el punto destino, y también de seleccionar la tabla de vuelo adecuada. La función *vaxActorFlyUpdateSklsroot()* será llamada solamente en el caso de que el actor esté dentro del frustum, y se encarga de hacer llamadas a las funciones *vaxActorKftableAdd()* y *vaxActorKftableRemove()* para cambiar la tabla de vuelo que el actor está ejecutando en cada momento.

9.5.5 Módulo de definición de l Actor virtual ejemplo y su "microAPI" de manejo.

En este apartado se va a describir el contenido de los ficheros "*birdVa.h*" y "*birdVa.c*" encargados de realizar la creación del actor virtual ejemplo, y de dotarlo de todas las capacidades que puedan ser necesarias en el programa final de aplicación.

Algunas de las características del actor virtual que se va a implementar son las siguientes:

- Los actores se crearan por clonación de un actor prototipo.
- Cada actor gestionará de forma interna sus mecanismos de vuelo, mirada, parpadeo, expresión y locución.
- Cada actor tendrá su propia personalidad, que afectará a la forma en la que se ejecutan los mecanismos anteriores (por ejemplo en la rapidez con la que parpadea), y también en su apariencia externa. Un actor podrá ser representado con cuatro posibles tipos de piel.
- Cada actor tendrá objetivos de vuelo y mirada totalmente diferentes, y en general, su comportamiento será totalmente autónomo.

Para ocultar la complejidad del actor se definirá un "*microAPI*" que proporcione funciones especificas para trabajar con actores de este tipo. Mediante este API se consigue el objetivo final de integrar actores virtuales en una aplicación de simulación ocultando totalmente la complejidad inherente a la creación y gestión de su estructura jerárquica.

Una vez finalizada la implementación del actor de tipo "*Bird*" en los ficheros "*birdVa.h*" y "*birdVa.c*", una tercera persona podrá personalizar el actor virtual de los siguientes modos:

- Sustituyendo los ficheros que definen la geometría del actor que están definidos empleando un criterio de nombre que será visto más adelante.
- Redefininendo las 4 texturas que definen los distintos tipos de piel de los actores.
- Personalizando las funciones de gestión de su comportamiento empleando para ello el API incluido en el fichero "*birdVa.h*",

El contenido del fichero de cabecera "*vaBird.h*" que contiene las funciones del "*microAPI*" para el manejo del actor de tipo *"Bird"* es el siguiente:

```
#ifndef __BIRD_VA_H__
#define __BIRD_VA_H__

extern vaActclass *birdVaActclassCreate(char *geometryPath);
extern vaActclass *birdVaGetActclass();

extern vaActor *birdVaCreate(float3 pos, float3 hpr, float normSpeed);

extern void birdVaSetBlinkAttention(vaActor *act, float attention);
extern void birdVaDoCheep(vaActor *act);
```

```
extern void birdVaSetCurPosHpr(        vaActor *act, float3 pos, float3 hpr);
extern void birdVaSetFlyTarget(        vaActor *act, float3 targetPos);
extern void birdVaSetHappiness(        vaActor *act, int happiness);
extern void birdVaSetSecondLookTarget( vaActor *act, float3 targetPos);
extern void birdVaSetActiveLookTarget( vaActor *act, int activeTarget);

extern int  birdVaIsWaiting(           vaActor *act);
extern int  birdVaGetFlyState(         vaActor *act);

extern void birdVaFlyUpdateRefpoint(   vaActor *act);

typedef void   (*birdBehaFunc)(        vaActor *act);
extern void birdVaSetPerFrameUpdateDofsFunc(          vaActclass *ac, birdBehaFunc func);
extern void birdVaSetPerFrameUpdateRefpointFunc(      vaActclass *ac, birdBehaFunc func);

#endif
```

Respecto al microAPI de gestión de los actores de la clase "*Bird*", su *vaActorClass* es creada mediante la función ***birdVaActclassCreate()***, esta función toma como parámetro el nombre de un directorio en el que se encuentran los ficheros con las geometrías que se necesitan para componer al actor, estos ficheros han de tener unos nombres concretos y la geometría que contienen han de presentar una orientación y dimensiones adecuadas. Los nombres de estos ficheros terminan con una letra mayúscula que sirven para diferenciar entre los distintos niveles de detalle, así para la construcción del pico que esta formado por tres niveles de detalle se necesitaran los ficheros de nombres *"geoBeakA.obj"*, *"geoBeakB.obj", "geoBeakC.obj"*.

Los ficheros de geometría necesarios para son especificados en la siguiente tabla:

body	(4) geoBodyA.obj, geoBodyB.obj, geoBodyC.obj y geoBodyD.obj.
wingL	(3) geoWingLA.obj, geoWingLB.obj y geoWingLC.obj.
wingR	(3) geoWingRA.obj, geoWingRB.obj y geoWingRC.obj.
wingtipL	(2) geoWingtipLA.obj y geoWingtipLB.obj.
wingtipR	(2) geoWingtipRA.obj y geoWingtipRB.obj.
tail	(2) geoTailA.obj y geoTailB.obj.
head	(3) geoHeadA.obj, geoHeadB.obj y geoHeadC.obj.
eyeL	(2) geoEyeLA.obj y geoEyeLB.obj.
eyeR	(2) geoEyeRA.obj y geoEyeRB.obj.
eyelidUL	(2) geoEyelidULA.obj y geoEyelidULB.obj.
eyelidUR	(2) geoEyelidURA.obj y geoEyelidURB.obj.
eyelidBL	(2) geoEyelidBLA.obj y geoEyelidBLB.obj.
eyelidBR	(2) geoEyelidBRA.obj y geoEyelidBRB.obj.
beak	(3) geoBeakA.obj, geoBeakB.obj y geoBeakC.obj.

El contenido de los ficheros de geometría actuales es el mostrado en el apartado 9.5.3. El aspecto externo de esta clase de actores puede cambiar totalmente con la simple modificación del contenido de estos ficheros.

La definición de la clase de actor se completa fijando las funciones que han de ser llamadas por frame para actualizar la posición de sus *Reference Points*, y también los valores de sus grados de libertad. Esto es hecho mediante las funciones ***birdVaSetPerFrameUpdateRefpointFunc()*** y ***birdVaSetPerFrameUpdateDofsFunc()***. La función ***birdVaFlyUpdateRefpoint()*** actualiza los valores del *refpoint* según el mecanismo de vuelo del actor, ha de ser llamada desde el interior de la función de control del *refpoint* pasada a la función *birdVaSetPerFrameUpdateRefpointFunc().* La función ***birdVaGetActclass()*** retorna un puntero a la clase de actor creada.

Respecto a las funciones de creación y gestión de los actores de tipo "*Bird*", el actor es creado en una posición y con una personalidad mediante la función ***birdVaCreate()***. Su posición y orientación inicial en el espacio son fijadas mediante ***birdVaSetCurPosHpr()***, el punto hacia el que intentara volar es indicado mediante ***birdVaSetFlyTarget()***, ese punto también actuar como objetivo principal de la mirada, es posible fijar un segundo objetivo de la mirada mediante la función ***birdVaSetSecondLookTarget()***. Mediante ***birdVaSetActiveLookTarget()*** se indica al actor si ha de mirar al objetivo principal o secundario. La función ***birdVaSetBlinkAttention()*** modula la forma en la que el actor parpadea (es un valor entre 0.0 y 1.0). El valor ***birdVaSetHappiness()*** permite fijar al actor un valor entre -1.0 y 1.0 que se emplea para modificar su expresión facial. La función ***birdVaDoCheep()*** fuerza al que el actor simule la emisión de un sonido, y por último, la función ***birdVaGetFlyState()*** permite consultar el estado de vuelo del actor, los valores retornados son los ya definidos en el fichero de extensión *"vaxFly.h"*

El fichero principal de definición del actor virtual se llama "*birdVa.c*", en él se encuentran implementadas las funciones que definen el microAPI que facilitan la comunicación con la clase "*Bird*", y también de funciones que se encargan de su creación.

La función principal de creación de la clase de actor de tipo "*Bird*" se llama ***birdActclassCreateBasis()***, en ella definen las estructuras *vaDof* y *vaSkelprot* necesarias, se rellenan con los datos adecuados, y se conectan entre ellas con el *vaActclass* para definir la estructura jerárquica de los actores de ese tipo. También se crean y asignan los datos de usuario que almacenara esa *vaActclass*, y se asignan las funciones de callback que gestionaran el comportamiento de los actores de esa clase.

La función ***birdActclassCreateExtensions()*** está encargada de configurar las extensiones necesarias para su trabajo con la clase de actor "*Bird*". Esta función obtiene los índices de los *vaDofs*, y también los índices de los puntos de articulación y los emplea para configurar las extensión de control de la mirada ("*vaxLook*"), control del parpadeo ("*vaxBlink*") y control del vuelo ("*vaxFly*"), previa a la configuración de estas extensiones empleadas directamente por los actores, se han de configurar las extensiones de bajo nivel "*vaxPose*" y "*vaxKftable*", las cuales serán empleadas en la definición de las extensiones "*vaxBlink*" para definir las posturas de los párpados abiertos y cerrados, y "*vaxFly*" para definir las tablas de *keyframing* que se corresponden con los distintos tipos de vuelo.

La función ***birdVaCreatePrototypeActor()*** carga los ficheros de geometría que definen a un actor especial que actuará como prototipo para la creación del resto de los de esa clase, y crea un también crea los nodos *vaLod* necesarios. Por último, define un actor completo listo para ser dibujado conectando de forma adecuada los *nodos de geometría* con los *nodos vaLod* y con los *nodos Skeleton y Actor*.

Las funciones *birdActclassCreateBasis()*, *birdActclassCreateExtensions()* y *birdVaCreatePrototypeActor()* son llamadas empleadas dentro de la función principal ***birdVaActclassCreate()*** para definir totalmente la clase de actor "*Bird*".

A continuación se mostrará el contenido del fichero "*birdVa.c*" en el que se encuentra la implementación de las anteriores funciones. Dicho fichero aparece fragmentado en varios bloques para facilitar el intercalado de comentarios:

```
#include "stdlib.h"
#include "stdio.h"

#include "vaApi.h"              //-Cabecera del API de actores virtuales.

#include "loadGeoFile.h"        //-Fichero en el que se definen las funciones:
                                //   loadGeometryFromFile()
                                //   loadMaterialFromFile() y
                                //   materialApplyPreDraw() y materialApplyPostDraw().

#include "vaxPose.h"            //-Ficheros en los que se definen las extensiones
#include "vaxBlink.h"           // que se emplearan en la definicion de los actores
#include "vaxLook.h"            // de tipo Bird.
#include "vaxKftable.h"
#include "vaxFly.h"

#include "birdVa.h"

#define BEAKSTATE_CLOSE         0       //-Estados del pico.
#define BEAKSTATE_OPENING       1
#define BEAKSTATE_CLOSING       2

#define EXPRSTATE_HAPPY         1       //-Expresiones faciales.
#define EXPRSTATE_NEUTRAL       0
#define EXPRSTATE_UNHAPPY      -1

static vaActclass *BirdActclass = NULL; //Puntero a la actorclass,
                                        //Sera utilizada internamente en este modulo.

//-Estructura que almacena los datos particulares de la clase de actor bird.
typedef struct birdAcUdataStruct{
   int     poseHappy;           //-Indices de las poses de las expresiones
   int     poseNeutral;         // faciales.
   int     poseUnhappy;

   int     poseBeakOpen;       //-Indices de las poses del pico.
   int     poseBeakClose;

   vaActor *protBird;           //-Actor que actua como prototipo para la creacion del resto de la clase.

   void * texA;                 //-Distintas texturas que pueden tener los
   void * texB;                 // actores de esta clase.
```

```
   void * texC;
   void * texD;

   float    curActorNormSpeed;          //-Velocidad normalizada de los actores a crear por esta clase.
   birdBehaFunc   behaFunc;             //-Funcion de gestion de su coportamiento.
}birdAcUdata;

//-Estructura que almacena los datos particulare de un actor de esta clase.
typedef struct birdUdataStruct{
   int    beakState;                    //-Parametros que controlan la locucion del actor.
   float  beakPeriod;
   float  beakStartTime;
   int    doCheepDirty;

   int    exprState;                    //-Expresion facial actual.
   int    newExprState;                 //-Expresion facial siguiente. (para cambios suaves).

   float  normSpeed;                    //-Velocidad normalizada del actor.

   void * tex;                          //-Textura de este actor.

   float3 lookSecondTarget;             //-Puntos objetivo de la mirada.
   int    lookActiveTarget;
}birdUdata;
```

Como se puede observar, se incluyen los ficheros "*vaApi.h*", que es el fichero de cabecera de la librería de gestión de actores, así como los ficheros de cabecera de todas las extensiones empleadas: *"vaxPose.h"*, *"vaxKftable.h"*, *"vaxLook.h"*, *"vaxBlink.h"*, y *"vaxFly.h"*. También el propio fichero de cabecera "*birdVa.h*".

En el fichero *"loadGeoFile.h"* se encuentran definidos los prototipos de la función *loadGeometryFromFile()*, que carga una geometría en formato *.obj* sobre un nodo de tipo geométrico del grafo de escena, la función *loadMaterialFromFile(),* que genera un material a partir de una imagen almacenada en un fichero *.rgb, y las funciones materialApplyPreDraw() y materialApplyPostDraw(),* que reciben como parámetro el material generado por la función anterior, y se emplean para modificar el material con el que se dibuja un subgrafo de escena.

Las estructuras ***birdAcUdata*** y ***birdUdata*** definen los datos de usuario con información propia de ese tipo de actor que será almacenada en las estructuras *vaActclass* y *vaActor*.

De forma adicional a los mecanismos de *control de la mirada, parpadeo y vuelo* implementados mediante la utilización de las estructuras *vaExtension,* los actores de esta clase presentan dos mecanismos de control adicionales que no son almacenados en ninguna extensión*, el mecanismo de locución* y *el mecanismo de expresión*. La información necesaria para implementar estos mecanismos se encuentra en el interior de estas dos estructuras.

Para la definición del *mecanismo de control de la expresión* es necesario haber definido tres poses que se correspondan con expresiones faciales de felicidad, indiferencia o infelicidad. Los índices de esas

posturas son almacenados en los campos ***poseHappy***, ***poseNeutral*** y ***poseUnhappy*** de la estructura *birdAcUdata*. Adicionalmente en la estructura *birdUdata* se almacenaran los campos ***exprState*** y ***newExprState*** que contienen un entero que hace referencia al estado de la expresión facial en un determinado momento (estos valores pueden ser *EXPRSTATE_HAPPY*, *EXPRSTATE_NEUTRAL* o *EXPRSTATE_UNHAPPY)*, y también, el valor al que va a cambiar en el futuro, este segundo campo es necesario para poder realizar transiciones suaves entre dos expresiones faciales,

Para la definición del *mecanismo de control de la locución* es necesario haber definido dos poses, que se corresponden con la posición de abierto y cerrado del pico. Los índices de estas posturas son almacenados en los campos ***poseBeakOpen*** y ***poseBeakClose*** de la estructura *birdAcUdata.* El mecanismo de control de la locución es gestionado como una sencilla máquina de estados que puede pasar por los estados de *BEAKSTATE_CLOSE*, BEAKSTATE_OPENING y BEAKSTATE_CLOSING, el estado actual en el que se encuentra dentro de cada actor es almacenado en la variable ***beakState*** de la estructura *birdUdata*. El campo ***beakPeriod*** indica el tiempo en segundos que emplea un actor concreto en ejecutar la secuencia de apertura y cierre del pico, y ***beakStartTime*** es una variable auxiliar que almacena el instante en el que el pico comienza a abrirse. Por último ***doCheepDirty*** es un flag que sirve para forzar a que el actor ejecute una secuencia de abrir y cerrar el pico en un momento determinado.

En este ejemplo, la personalidad de cada actor es definida por medio de un único parámetro que se almacena en el *birdUdata* con el nombre de ***normSpeed***. Éste es un valor entre 0.0 y 1.0 que afecta a la rapidez con la que un determinado actor ejecuta sus movimientos, un valor de 0.0 se correspondería con un actor muy tranquilo, y un valor de 1.0, con un actor muy inquieto. Este valor afecta a la velocidad con la que se desplaza, a la forma en la que parpadea, a la rapidez con la que mira a los sitios, y también a su piel. El valor ***curActorNormSpeed*** almacenado en el *birdAcUdata*, hace referencia a también a este parámetro, almacenando el valor que será asignado al campo los *normSpeed* actores creados por esta clase de actor, este valor puede ir cambiando para generar actores de distintas personalidades.

La personalidad del actor (definida a través del parámetro único *normSpeed*) afecta al material con el que se dibuja, pudiendo presentar cuatro tipos distintos de materiales, estos materiales son almacenados en los campos ***texA***, ***texB***, ***texC*** y ***texD*** del *birdAcUdata*, la clase *birdUdata* almacena en su campo ***tex*** el puntero a la textura que ese actor emplea.

Las cuatro texturas que serán empleadas este ejemplo se almacenarán en los ficheros de nombres "*birdTopA.rgb*", "*birdTopB.rgb*", "*birdTopC.rgb*" y "*birdTopD.rgb*". El contenido actual de estos ficheros es el siguiente:

Figura 9-8. Contenido de los ficheros de imagen empleados para definir la piel del actor.

El campo ***protBird*** de la estructura *birdAcUdata* es un puntero al actor que actuará como prototipo para la creación del resto de los actores por clonación. Por último, el campo ***behaFunc*** es un puntero a una función en la que el usuario de la aplicación podrá emplear el micro-API de este tipo de actores para definir la forma en la que quiere que se comporte.

En el siguiente bloque de código se muestra el contenido de las funciones *birdActorUdataCreate()* y *birdActorUdataDelete()*, encargadas de crear y eliminar las estructuras *birdUdata* almacenada en cada actor, también el contenido de la función *birdComputeDofsFunc()* que es llamada para actualizar el estado interno del actor actuando sobre sus datos de usuario, y sobre las extensiones. Por último, se muestran las funciones *actorMaterialPreDrawFunc()* y *actorMaterialPostDrawFunc()*, empleadas para poder cambiar las propiedades del material de cada actor. Estas cinco funciones serán empleadas para la definición de la clase de actor en la función *birdActclassCreateBasis()*.

```
//-Funcion de creacion de los datos personalizados del actor de tipo  bird.
static void * birdActorUdataCreate(vaActor *act){
   float3 target;
   target[0]= target[1]= target[2] = 0.0f;

   vaxActorFlySetTarget(                act, target);
   vaxActorLookSetFirstTarget(          act, target);
   vaxActorLookSetActiveTarget(         act, 1);

   birdUdata   *b     = (birdUdata *)vaSgMalloc(sizeof(birdUdata));
   vaActclass  *ac   = vaActorGetActclass(act);
   birdAcUdata *bc = (birdAcUdata *)vaActclassGetBehaviourData(ac);

   //-El campo normSpeed determinara la personalidad del actor
   // en varios aspectos tales como la velocidad a la que se desplaza
   // la rapidez con la que parpadea o habla.
   b->normSpeed    =           bc->curActorNormSpeed;

   float birdSpeed = b->normSpeed;

   //-- set fly properties ------------------------------------//
   float vHor   = 2.0f +birdSpeed*6.0f;
   float vVer   = vHor/2.0f;
   float vTurn = 40.0f +birdSpeed*140.0f;
   vaxActorFlySetSpeeds(act, vHor, vVer, vTurn);

   //-- set lookAt properties ---------------------------------//
   float headSpeed =  10.0f +birdSpeed*200.0f;
   float eyesSpeed = 100.0f +birdSpeed*500.0f;
   vaxActorLookSetSpeeds( act,          headSpeed, eyesSpeed);

   //-- set Blink properties ----------------------------------//
   float openTimeMin = 0.1f+(1.0f-birdSpeed)*0.2f;
   float openTimeMax = 0.3f+(1.0f-birdSpeed)*0.3f;
   float waitTimeMin = 0.1f+(1.0f-birdSpeed)*0.2f;
   float waitTimeMax = 1.0f+(1.0f-birdSpeed)*2.0f;
   vaxActorBlinkSetSpeeds(act, openTimeMin, openTimeMax,  waitTimeMin, waitTimeMax);

   //-- set Speech Properties ---------------------------------//
   b->beakPeriod      = 0.1f +(1.0f-birdSpeed)*0.8f;
   b->beakState        = BEAKSTATE_CLOSE;
   b->beakStartTime  = vaGetCurTime();
   b->doCheepDirty   = 0;

   //-- set Skin texture --------------------------------------//
   if (b->normSpeed <= 0.25f)      b->tex = bc->texD;
   else if (b->normSpeed <= 0.50f) b->tex = bc->texC;
   else if (b->normSpeed <= 0.70f) b->tex = bc->texB;
   else                            b->tex = bc->texA;

   //-- set Expresion Properties ------------------------------//
   b->exprState     = EXPRSTATE_NEUTRAL;
   return b;
}

//-Funcion de borrado de los datos personalizados del actor de tipo  bird.
//
static void birdActorUdataDelete(void *actBehaData){
   vaSgFree(actBehaData);
}
```

```
//-Funcion de calculo de los posiciones de los DOFs del Actor y otras
// variables que definen su estado actual.
//
static void  birdComputeDofsFunc(vaActor *act){
   vaActclass *ac    = vaActorGetActclass(act);
   birdAcUdata *bc = (birdAcUdata *)vaActclassGetBehaviourData(ac);
   birdUdata   *b     = (birdUdata   *)vaActorGetBehaviourData(act);

   bc->behaFunc(act);

   //-- fijacion de los puntos objetivo de la mirada ----------//
   vaxActorLookSetSecondTarget( act, b->lookSecondTarget );
   vaxActorLookSetActiveTarget( act,   b->lookActiveTarget );

   //-- actuacion sobre la expresion ---------------------------------//
   switch (b->newExprState){
        case EXPRSTATE_HAPPY:
          if (b->exprState != EXPRSTATE_HAPPY)
             vaxActorPoseApply(act, bc->poseHappy, 5.0f);
          break;
        case EXPRSTATE_NEUTRAL:
          if (b->exprState != EXPRSTATE_NEUTRAL)
             vaxActorPoseApply(act, bc->poseNeutral, 5.0f);
          break;
        case EXPRSTATE_UNHAPPY:
          if (b->exprState != EXPRSTATE_UNHAPPY)
             vaxActorPoseApply(act, bc->poseUnhappy, 5.0f);
          break;
        }
   b->exprState = b->newExprState;

   //-- actuacion sobre el pico -------------------------------------//
   switch (b->beakState){
        case BEAKSTATE_CLOSE:
           if (vaGetCurTime() -b->beakStartTime > b->beakPeriod/2.0f){
             if (b->doCheepDirty || (rand()%40 == 0)){
                vaxActorPoseApply(act, bc->poseBeakOpen, b->beakPeriod/2);
                b->beakState       = BEAKSTATE_OPENING;
                b->beakStartTime = vaGetCurTime();
                b->doCheepDirty  = 0;
             }
          }
          break;
        case BEAKSTATE_OPENING:
           if (vaGetCurTime() - b->beakStartTime > b->beakPeriod/2){
             b->beakState = BEAKSTATE_CLOSING;
             vaxActorPoseApply(act, bc->poseBeakClose,  b->beakPeriod/2);
          }
            break;
        case BEAKSTATE_CLOSING:
           if (vaGetCurTime() - b->beakStartTime > b->beakPeriod){
             b->beakState = BEAKSTATE_CLOSE;
          }
            break;
   }
}

//-Funciones que se llaman al inicio y al final del dibujado de un actor
// para poder modificar las propiedades del material que se dibuja.
//
static void actorMaterialPreDrawFunc(vaActor *act){
```

```
    birdUdata *b = (birdUdata *)vaActorGetBehaviourData(act);
    materialApplyPreDraw(b->tex);
}

static void actorMaterialPostDrawFunc(vaActor *act){
    birdUdata *b = (birdUdata *)vaActorGetBehaviourData(act);
    materialApplyPostDraw(b->tex);
}
```

En la función ***birdActorUdataCreate()*** se inicializan los mecanismos de vuelo (extensión *vaxFly*), control de la mirada (extensión *vaxLook*), parpadeo (extensión *vaxBlink*) y también los mecanismos internos de locución y expresión. Como se puede observar la forma de comportamiento de estos mecanismos se ve afectada por el valor *normSpeed* que caracteriza la personalidad del actor. También se puede observar como este parámetro es empleado para seleccionar la textura de la piel del actor.

La función ***birdActorUdataDelete()*** simplemente libera la memoria creada por la función anterior.

La función ***birdComputeDofsFunc()*** es llamada automáticamente por la librería de gestión de actores en el caso del actor esté dentro de la pirámide de visión. En ella se encuentra el código encargado de actualizar el estado interno del actor y modificar los parámetros necesarios para que el actor se dibuje de una forma coherente. Lo primero que hace ésta, es llamar a la función *behaFunc* de la estructura *birdAcUdata*, en esta función el programa de aplicación podrá definir la forma en la que se quiere que se comporte el actor. A continuacón se realiza una actualización del estado de los mecanismos de control de expresión y locución, se puede observar como ambos mecanismo emplean como base a la extensión de gestión de posturas *vaxPose* para fijar de una forma cómoda las expresiones faciales o la postura del pico.

Las funciones ***actorMaterialPreDrawFunc()*** y ***actorMaterialPostDrawFunc()*** serán aplicadas en los *callback* de dibujado del nodo Actor para poder cambiar la textura de la piel de los actores, internamente emplean las funciones *materialApplyPreDraw()* y *materialApplyPreDraw()* definidas en el fichero auxiliar "*loadGeoFile.h*"

En el siguiente bloque se muestra el contenido de la función *birdActclassCreateBasis()*, encargada de definir los aspectos básicos de la clase de actor empleada para este ejemplo:

```
static vaActclass * birdActclassCreateBasis(char *geometryPath){
  //-- Creacion de los DOFs. ------------------------------------------------//
   vaDof *dofHeadAzim        = vaDofNew("headAzim");
   vaDof *dofHeadElev        = vaDofNew("headElev");
   vaDof *dofHeadTwist       = vaDofNew("headTwist");

   vaDof *dofWingRElev       = vaDofNew("wingRElev");
   vaDof *dofWingLElev       = vaDofNew("wingLElev");

   vaDof *dofWingtipR        = vaDofNew("wingtipRElev");
   vaDof *dofWingtipL        = vaDofNew("wingtipLElev");

   vaDof *dofBeakOpen        = vaDofNew("beakOpen");

   vaDof *dofTailElev        = vaDofNew("tailElev");
   vaDof *dofTailTwist       = vaDofNew("tailTwist");

   vaDof *dofEyeLAzim        = vaDofNew("eyeLAzim");
   vaDof *dofEyeLElev        = vaDofNew("eyeLElev");
   vaDof *dofEyeRAzim        = vaDofNew("eyeRAzim");
   vaDof *dofEyeRElev        = vaDofNew("eyeRElev");

   vaDof *dofEyelidULOpen  = vaDofNew("eyelidULOpen");
   vaDof *dofEyelidULExpr  = vaDofNew("eyelidULExpr");
   vaDof *dofEyelidUROpen  = vaDofNew("eyelidUROpen");
   vaDof *dofEyelidURExpr   = vaDofNew("eyelidURExpr");

   vaDof *dofEyelidDLOpen  = vaDofNew("eyelidDLOpen");
   vaDof *dofEyelidDROpen  = vaDofNew("eyelidDROpen");

   vaDofSetMinMax( dofHeadAzim                    , -40.0f, 40.0f );

   vaDofSetMinMax( dofHeadElev       , -70.0f, 30.0f );
   vaDofSetMinMax( dofHeadTwist      , -60.0f, 60.0f );

   vaDofSetMinMax( dofWingRElev      , -60.0f, 60.0f );
   vaDofSetMinMax( dofWingLElev      , -60.0f, 60.0f );
   vaDofSetMinMax( dofWingtipR       , -60.0f, 60.0f );
   vaDofSetMinMax( dofWingtipL       , -60.0f, 60.0f );
   vaDofSetMinMax( dofBeakOpen       , -60.0f,  0.0f );
   vaDofSetMinMax( dofTailElev       , -90.0f ,50.0f );

   vaDofSetMinMax( dofEyeLAzim       , -70.0f, 70.0f );
   vaDofSetMinMax( dofEyeLElev       , -70.0f, 70.0f );
   vaDofSetMinMax( dofEyeRAzim       , -70.0f, 70.0f );
   vaDofSetMinMax( dofEyeRElev       , -70.0f, 70.0f );

   vaDofSetMinMax( dofEyelidULOpen , -80.0f,  0.0f );
   vaDofSetMinMax( dofEyelidULExpr , -80.0f, 80.0f );
   vaDofSetMinMax( dofEyelidUROpen, -80.0f,  0.0f );
   vaDofSetMinMax( dofEyelidURExpr , -80.0f, 80.0f );
   vaDofSetMinMax( dofEyelidDLOpen ,   0.0f, 80.0f );
   vaDofSetMinMax( dofEyelidDROpen,   0.0f, 80.0f );

   //-- Creacion de los SkelProts.-------------------------------------------//
   vaSkelprot *skpWingL      = vaSkelprotNew("wingL");
   vaSkelprot *skpWingR      = vaSkelprotNew("wingR");
   vaSkelprot *skpWingtipL   = vaSkelprotNew("wingtipL");
   vaSkelprot *skpWingtipR   = vaSkelprotNew("wingtipR");
```

```
vaSkelprot *skpTail      = vaSkelprotNew("tail");
vaSkelprot *skpHead      = vaSkelprotNew("head");
vaSkelprot *skpEyeL      = vaSkelprotNew("eyeL");
vaSkelprot *skpEyeR      = vaSkelprotNew("eyeR");
vaSkelprot *skpEyelidUL  = vaSkelprotNew("eyelidUL");
vaSkelprot *skpEyelidUR  = vaSkelprotNew("eyelidUR");
vaSkelprot *skpEyelidDL  = vaSkelprotNew("eyelidDL");
vaSkelprot *skpEyelidDR  = vaSkelprotNew("eyelidDR");
vaSkelprot *skpBeak      = vaSkelprotNew("beak");

//-- Rellenado del contenido de los SkelProts. ----------------------------//
vaSkelprotSetOffsetPos(  skpWingL, 0.1f, 0.2f, 0.0f);
vaSkelprotSetOffsetHpr(  skpWingL, 0.0f, 0.0f, 90.0f);
vaSkelprotSetRyDOF(      skpWingL, dofWingLElev);

vaSkelprotSetOffsetPos(  skpWingR, 0.1f,  -0.2f,  0.0f);
vaSkelprotSetOffsetHpr(  skpWingR, 0.0f,   0.0f, -90.0f);
vaSkelprotSetRyDOF(      skpWingR, dofWingRElev);

vaSkelprotSetOffsetPos(  skpWingtipL, 0.77f, 0.0f, 0.0f);
vaSkelprotSetOffsetHpr(  skpWingtipL, 0.0f,  0.0f, 0.0f);
vaSkelprotSetRyDOF(      skpWingtipL, dofWingtipL);

vaSkelprotSetOffsetPos(  skpWingtipR, 0.77f, 0.0f, 0.0f);
vaSkelprotSetOffsetHpr(  skpWingtipR, 0.0f,  0.0f, 0.0f);
vaSkelprotSetRyDOF(      skpWingtipR, dofWingtipR);

vaSkelprotSetOffsetPos(  skpTail, -0.7f, 0.0f, 0.0f);
vaSkelprotSetOffsetHpr(  skpTail, 0.0f, 0.0f, 180.0f);
vaSkelprotSetRyDOF(      skpTail, dofTailElev);
vaSkelprotSetRxDOF(      skpTail, dofTailTwist);

vaSkelprotSetOffsetPos(  skpHead, 0.5f, 0.0f, 0.0f);
vaSkelprotSetOffsetHpr(  skpHead, 0.0f, 0.0f, 0.0f);
vaSkelprotSetRzDOF(      skpHead, dofHeadAzim);
vaSkelprotSetRyDOF(      skpHead, dofHeadElev);
vaSkelprotSetRxDOF(      skpHead, dofHeadTwist);

vaSkelprotSetOffsetPos(  skpEyeL, 0.15f, -0.15f, 0.14f);
vaSkelprotSetOffsetHpr(  skpEyeL, 0.0f, 0.0f, 0.0f);
vaSkelprotSetRzDOF(      skpEyeL, dofEyeLAzim);
vaSkelprotSetRyDOF(      skpEyeL, dofEyeLElev);

vaSkelprotSetOffsetPos(  skpEyeR, 0.15f, 0.15f, 0.14f);
vaSkelprotSetOffsetHpr(  skpEyeR, 0.0f, 0.0f, 0.0f);
vaSkelprotSetRzDOF(      skpEyeR, dofEyeRAzim);
vaSkelprotSetRyDOF(      skpEyeR, dofEyeRElev);

vaSkelprotSetOffsetPos(  skpEyelidUL, 0.15f, -0.15f, 0.14f);
vaSkelprotSetOffsetHpr(  skpEyelidUL, 0.0f, -90.0f, 0.0f);
vaSkelprotSetRyDOF(      skpEyelidUL, dofEyelidULOpen);
vaSkelprotSetRxDOF(      skpEyelidUL, dofEyelidULExpr);

vaSkelprotSetOffsetPos(  skpEyelidUR, 0.15f, 0.15f, 0.14f);
vaSkelprotSetOffsetHpr(  skpEyelidUR, 0.0f, -90.0f, 0.0f);
vaSkelprotSetRyDOF(      skpEyelidUR, dofEyelidUROpen);
vaSkelprotSetRxDOF(      skpEyelidUR, dofEyelidURExpr);

vaSkelprotSetOffsetPos(  skpEyelidDL, 0.15f,  -0.15f, 0.14f);
vaSkelprotSetOffsetHpr(  skpEyelidDL, 0.0f, 90.0f, 0.0f);
vaSkelprotSetRyDOF(      skpEyelidDL, dofEyelidDLOpen);
```

```
vaSkelprotSetOffsetPos(   skpEyelidDR, 0.15f,  0.15f, 0.14f);
vaSkelprotSetOffsetHpr(   skpEyelidDR, 0.0f, 90.0f, 0.0f);
vaSkelprotSetRyDOF(       skpEyelidDR, dofEyelidDROpen);

vaSkelprotSetOffsetPos(   skpBeak, 0.15f, 0.0f, 0.0f);
vaSkelprotSetOffsetHpr(   skpBeak, 0.0f, 0.0f, 0.0f);
vaSkelprotSetRyDOF(       skpBeak, dofBeakOpen);

//-- Definicion de la topologia conectando los skelprots. ----------//
vaSkelprotAddChild(skpWingL,       skpWingtipL);
vaSkelprotAddChild(skpWingR,       skpWingtipR);
vaSkelprotAddChild(skpHead,         skpEyeL);
vaSkelprotAddChild(skpHead,        skpEyeR);
vaSkelprotAddChild(skpHead,        skpEyelidUL);
vaSkelprotAddChild(skpHead,        skpEyelidUR);
vaSkelprotAddChild(skpHead,        skpEyelidDL);
vaSkelprotAddChild(skpHead,        skpEyelidDR);
vaSkelprotAddChild(skpHead,        skpBeak);

//-- Se fijan las distancias de desactivacion de los nodos Skeleton ------//
//-LODs 5, 20, 100, 500, 2500
vaSkelprotSetOffDist( skpHead,      500.0f);
vaSkelprotSetOffDist( skpEyeL,       20.0f);
vaSkelprotSetOffDist( skpEyeR,       20.0f);
vaSkelprotSetOffDist( skpEyelidUL,  20.0f);
vaSkelprotSetOffDist( skpEyelidDL,  20.0f);
vaSkelprotSetOffDist( skpEyelidUR,  20.0f);
vaSkelprotSetOffDist( skpEyelidDR,  20.0f);
vaSkelprotSetOffDist( skpBeak,      100.0f);
vaSkelprotSetOffDist( skpTail,      100.0f);
vaSkelprotSetOffDist( skpWingL,     500.0f);
vaSkelprotSetOffDist( skpWingR,     500.0f);
vaSkelprotSetOffDist( skpWingtipL,  100.0f);
vaSkelprotSetOffDist( skpWingtipR, 100.0f);

//-- Creacion de la ActorClass y conexion de los skelprot raices ---------//
vaActclass *birdClass = vaActclassNew("sampleBirdClass");

vaActclassAddRootSkelprot(birdClass,  skpWingL);
vaActclassAddRootSkelprot(birdClass,  skpWingR);
vaActclassAddRootSkelprot(birdClass,  skpTail);
vaActclassAddRootSkelprot(birdClass,  skpHead);

//-- Generacion de la tabla de indices de vaDofs y vaSkelprots-----------//
vaActclassCompile( birdClass);

//-- Impresion para validar la correccion la vaActclass creada  ----------//
vaActclassPrint(   birdClass);

//--- Adicion de los datos de usuario y de las funciones de control. -----//
birdAcUdata *bcUdata   = (birdAcUdata *)vaSgMalloc(sizeof(birdAcUdata));

bcUdata->texA = loadMaterialFromFile(geometryPath, "./birdTopA.rgb");
bcUdata->texB = loadMaterialFromFile(geometryPath, "./birdTopB.rgb");
bcUdata->texC = loadMaterialFromFile(geometryPath, "./birdTopC.rgb");
bcUdata->texD = loadMaterialFromFile(geometryPath, "./birdTopD.rgb");

float4 bsph;
bsph[0] = bsph[1] = bsph[2] = 0.0f;
bsph[3] = 1.5f;
```

```
    vaActclassSetSklsrootBsphere( birdClass, bsph);

    vaActclassSetBehaviourData(                 birdClass, bcUdata);

    vaActclassSetActorBehaDataCreateFunc(       birdClass, birdActorUdataCreate);
    vaActclassSetActorBehaDataDeleteFunc(       birdClass, birdActorUdataDelete);

    vaActclassSetDofsBehaviourFunc(             birdClass, birdComputeDofsFunc);
    vaActclassSetSklsrootBehaviourFunc(         birdClass, vaxActorFlyUpdateSklsroot);

    vaActclassSetDrawFuncs(                     birdClass, actorMaterialPreDrawFunc, actorMaterialPostDrawFunc);
    return birdClass;
}
```

La función ***birdActclassCreateBasis()*** retorna un puntero a una estructura vaActclass que contiene la definición básica del tipo de actor que se va a emplear en este ejemplo. La función comienza creando todas las estructuras *vaDof* que serán necesarias. Los nombres asignados a estos grados de libertad serán empleados posteriormente por la función *vaActclassFindDofIndexByName()* para obtener el índice de ese *dof* dentro de la clase de actor. Mediante el comando *vaDofSetMinMax()* se fijan márgenes en los que se puede mover cada grado de libertad. A continuación se crean todas las estructuras de tipo *vaSkelprot* que serán necesarias, asignándoles un nombre adecuado, esos nombres serán empleados posteriormente para obtener los índices de estos puntos de articulación mediante la función *vaActclassFindSklIndexByName()*. A continuación se definen los offsets de traslación y rotación de cada uno de estos puntos de articulación, y se vinculan con las estructuras *vaDof* correspondientes. Se continúa describiendo la topología del actor conectando las estructuras *vaSkelprot* entre sí mediante el uso de la función *vaSkelprotAddChild()*, y con el vaActclass mediante la función *vaActclassAddRootSkelprot()*. Se fijan las distancias de desactivación de los puntos de articulación, y se llama a la función ***vaActclassCompile()*** que se encarga de comprobar que todas las conexiones entre las estructuras *vaActclass*, *vaSkelprot* y *vaDof* son coherentes y generar la lista de índices de los grados de libertad y la lista de índices de los puntos de articulación. La función ***vaActclassPrint()*** imprime en un shell información sobre la organización topológica de la clase de actor para comprobar que todo ha sido generado de la forma deseada. A continuación se reserva memoria para estructura de tipo *birdAcUdata* que almacena datos de usuario de esta clase de actor, y se rellenan sus campos *texA*, *texB*, *texC* y *texD* con punteros a las estructuras proporcionadas por la función *loadMaterialFromFile()*, este fichero emplea los ficheros de textura de nombres "*birdTopA.rgb*", "*birdTopB.rgb*", "*birdTopC.rgb*" y "*birdTopD.rgb*" que se han de encontrar en el directorio indicado por la variable *geometryPath* pasada como parámetro. Se continúa con la asignación de los datos de usuario a la *vaActclass* mediante la función *vaActclassSetBehaviourData(),* también se asigna la *Sklsroot Bsphere.* Y por último se asignan las rutinas de *callback* que serán empleadas en la creación y eliminación de los datos de usuario del *vaActor -birdActorUdataCreate()* y *birdActorUdataDelete()-,* la función de actualización de la posición y orientación del *Skeletons Root* del *Actor* -en este caso se emplea directamente la función *vaxActorFlyUpdateSklsroot()* perteneciente a la extensión *vaxFly*-, la función de actualización de los *dofs* del actor -*birdComputeDofsFunc()*- y las funciones de modificación del material del actor - *actorMaterialPreDrawFunc()* y *actorMaterialPreDrawFunc()*-.

En el bloque que sigue se muestra el contenido de la función *birdActclassCreateExtensions()*, encargada de inicializar los mecanismos de control de vuelo, parpadeo y control de la mirada, así como de inicializar los datos que se emplearan en los mecanismos de control de la expresión y locución.

```
static void birdActclassCreateExtensions(vaActclass *ac){
   //-- Obtencion de los indices de los nodos Skeleton ----------------------//
   int headIndex              = vaActclassFindSklIndexByName(ac, "head");
   int lEyeIndex              = vaActclassFindSklIndexByName(ac, "eyeL");
   int rEyeIndex              = vaActclassFindSklIndexByName(ac, "eyeR");
   int ulEyelidIndex          = vaActclassFindSklIndexByName(ac, "eyelidUL");
   int blEyelidIndex          = vaActclassFindSklIndexByName(ac, "eyelidDL");
   int urEyelidIndex          = vaActclassFindSklIndexByName(ac, "eyelidUR");
   int brEyelidIndex          = vaActclassFindSklIndexByName(ac, "eyelidDR");
   int bBeakIndex             = vaActclassFindSklIndexByName(ac, "beak");
   int tailIndex              = vaActclassFindSklIndexByName(ac, "tail");
   int lWingIndex             = vaActclassFindSklIndexByName(ac, "wingL");
   int rWingIndex             = vaActclassFindSklIndexByName(ac, "wingR");
   int lWingtipIndex          = vaActclassFindSklIndexByName(ac, "wingtipL");
   int rWingtipIndex          = vaActclassFindSklIndexByName(ac, "wingtipR");

   //-- Obtencion de los indices de los grados de libertad. ----------------//
   int headTwistIndex         = vaActclassFindDofIndexByName(ac, "headTwist");
   int headAzimIndex          = vaActclassFindDofIndexByName(ac, "headAzim");
   int headElevIndex          = vaActclassFindDofIndexByName(ac, "headElev");
   int LeyeAzimIndex          = vaActclassFindDofIndexByName(ac, "eyeLAzim");
   int LeyeElevIndex          = vaActclassFindDofIndexByName(ac, "eyeLElev");
   int ReyeAzimIndex          = vaActclassFindDofIndexByName(ac, "eyeRAzim");
   int ReyeElevIndex          = vaActclassFindDofIndexByName(ac, "eyeRElev");
   int UReyelidElevIndex      = vaActclassFindDofIndexByName(ac, "eyelidUROpen");
   int BReyelidElevIndex      = vaActclassFindDofIndexByName(ac, "eyelidDROpen");
   int ULeyelidElevIndex      = vaActclassFindDofIndexByName(ac, "eyelidULOpen");
   int BLeyelidElevIndex      = vaActclassFindDofIndexByName(ac, "eyelidDLOpen");
   int UReyelidExprIndex      = vaActclassFindDofIndexByName(ac, "eyelidURExpr");
   int ULeyelidExprIndex      = vaActclassFindDofIndexByName(ac, "eyelidULExpr");
   int BbeakOpenIndex         = vaActclassFindDofIndexByName(ac, "beakOpen");
   int RwingTurnIndex         = vaActclassFindDofIndexByName(ac, "wingRElev");
   int RwingtipTurnIndex      = vaActclassFindDofIndexByName(ac, "wingtipRElev");
   int LwingTurnIndex         = vaActclassFindDofIndexByName(ac, "wingLElev");
   int LwingtipTurnIndex      = vaActclassFindDofIndexByName(ac, "wingtipLElev");
   int TailTurnIndex          = vaActclassFindDofIndexByName(ac, "tailElev");

   //-- Obtencion de los limites de algunos grados de libertad ------------//
   float headAzimMax, headAzimMin, headElevMax, headElevMin;
   float eyesAzimMax, eyesAzimMin, eyesElevMax, eyesElevMin;
   vaActclassGetDofBounds(ac,  headAzimIndex, &headAzimMin, &headAzimMax);
   vaActclassGetDofBounds(ac,  headElevIndex, &headElevMin, &headElevMax);
   vaActclassGetDofBounds(ac,  LeyeAzimIndex, &eyesAzimMin, &eyesAzimMax);
   vaActclassGetDofBounds(ac,  LeyeElevIndex, &eyesElevMin, &eyesElevMax);

   //--- Adicion del extension slot de lookAat ----------------------------//
   vaExtension *lookExt = vaxLookInit();
   vaActclassAddExtension( ac, lookExt, -1);
   vaxActclassLookSetDofsLimits(      ac, headAzimMin, headAzimMax, headElevMin, headElevMax,
                                      eyesAzimMin, eyesAzimMax, eyesElevMin, eyesElevMax);

   vaxActclassLookSetSklIndexes(      ac, headIndex, lEyeIndex, rEyeIndex);
   vaxActclassLookSetDofIndexes(      ac, headAzimIndex, headElevIndex, headTwistIndex,
                                             LeyeAzimIndex, LeyeElevIndex, ReyeAzimIndex, ReyeElevIndex);

   vaxActclassLookSetLodDistances(ac, 5.0f, 20.0f, 100.f, 500.f);
```

```
//--- Adicion del extension slot de poses ----------------------------//
vaExtension *poseExt       = vaxPoseInit();
vaActclassAddExtension(ac, poseExt, -1);

int numDofs                = 4;
int   *dofIndexes          = (int  *)vaSgMalloc(numDofs*sizeof(int  ));
float *dofValues           = (float*)vaSgMalloc(numDofs*sizeof(float));

dofIndexes[0]              = UReyelidExprIndex;
dofIndexes[1]              = ULeyelidExprIndex;
dofValues[0]               =  45.0f;
dofValues[1]               = -45.0f;
int poseExprHappy          = vaxActclassPoseCreate(ac, 2, dofIndexes, dofValues);
dofValues[0]               = -45.0f;
dofValues[1]               =  45.0f;
int poseExprUnhappy        = vaxActclassPoseCreate(ac, 2, dofIndexes, dofValues);
dofValues[0]               =  10.0f;
dofValues[1]               = -10.0f;
int poseExprNeutral        = vaxActclassPoseCreate(ac, 2, dofIndexes, dofValues);

dofIndexes[0]              = ULeyelidElevIndex;
dofIndexes[1]              = UReyelidElevIndex;
dofIndexes[2]              = BLeyelidElevIndex;
dofIndexes[3]              = BReyelidElevIndex;
dofValues[0]               =  30.0f;
dofValues[1]               =  30.0f;
dofValues[2]               = -10.0f;
dofValues[3]               = -10.0f;
int poseOpenEyes           = vaxActclassPoseCreate(ac, 4, dofIndexes, dofValues);
dofValues[0]               =  90.0f;
dofValues[1]               =  90.0f;
dofValues[2]               = -90.0f;
dofValues[3]               = -90.0f;
int poseCloseEyes          = vaxActclassPoseCreate(ac, 4, dofIndexes, dofValues);

dofIndexes[0]              = BbeakOpenIndex;
dofValues[0]               = 0.0f;
int poseBeakClose           = vaxActclassPoseCreate(ac, 1, dofIndexes, dofValues);
dofValues[0]               = 30.0f;
int poseBeakOpen           = vaxActclassPoseCreate(ac, 1, dofIndexes, dofValues);

vaSgFree(dofIndexes);
vaSgFree(dofValues);


//--- Adicion del extension slot de blink ----------------------------------//
vaExtension *blinkExt      = vaxBlinkInit();
vaActclassAddExtension(ac, blinkExt, -1);

vaxActclassBlinkSetPoseIndexes( ac, poseOpenEyes, poseCloseEyes);
vaxActclassBlinkSetLodDistance( ac, 20.0f);

//--- Adicion del extension slot de keyframing ---------------------------//
vaExtension *kftableExt = vaxKftableInit();
vaActclassAddExtension(ac, kftableExt, -1);

int    numKfDofs           = 6;
int    numKfTimes          = 3;
int   *kfDofIndexes        = (int *)vaSgMalloc(numDofs *sizeof(int  ));
float *kfTimes             = (float*)vaSgMalloc(numKfTimes*sizeof(float));
```

```
float *values                    = (float*)vaSgMalloc(numKfDofs*numKfTimes*sizeof(float));

kfDofIndexes[0]  = RwingTurnIndex;
kfDofIndexes[1]  = RwingtipTurnIndex;
kfDofIndexes[2]  = LwingTurnIndex;
kfDofIndexes[3]  = LwingtipTurnIndex;
kfDofIndexes[4]  = TailTurnIndex;
kfDofIndexes[5]  = VADOF_SKLSROOT_PZ;

//-- kfDofValues Up.--//
values[+0]=  75.0f;  values[1*6 +0]= -25.0f;  values[2*6 +0]=  75.0f;
values[+1]= -45.0f;  values[1*6 +1]=  45.0f;  values[2*6 +1]= -45.0f;
values[+2]=  75.0f;  values[1*6 +2]= -25.0f;  values[2*6 +2]=  75.0f;
values[+3]= -45.0f;  values[1*6 +3]=  45.0f;  values[2*6 +3]= -45.0f;
values[+4]= -45.0f;  values[1*6 +4]=  45.0f;  values[2*6 +4]= -45.0f;
values[+5]=   0.1f;  values[1*6 +5]=  -0.1f;  values[2*6 +5]=   0.1f;
kfTimes[0]= 0.0f;
kfTimes[1]= 0.3f;
kfTimes[2]= 0.4f;
int flyUpKftable    = vaxActclassKftableCreate(ac, numKfDofs, numKfTimes, kfDofIndexes, kfTimes, values,
    500.0f);

//-- kfDofValues Down. --/
values[+0]=  00.0f; values[1*6 +0]= -75.0f; values[2*6 +0]=  00.0f;
values[+1]= -00.0f; values[1*6 +1]=  75.0f; values[2*6 +1]= -00.0f;
values[+2]=  00.0f; values[1*6 +2]= -75.0f; values[2*6 +2]=  00.0f;
values[+3]= -00.0f; values[1*6 +3]=  75.0f; values[2*6 +3]= -00.0f;
values[+4]= -45.0f; values[1*6 +4]=  45.0f; values[2*6 +4]= -45.0f;
values[+5]=   0.0f; values[1*6 +5]=   0.0f; values[2*6 +5]=   0.0f;
kfTimes[0]= 0.0f;
kfTimes[1]= 0.6f;
kfTimes[2]= 1.5f;
int flyDownKftable  = vaxActclassKftableCreate(ac, numKfDofs, numKfTimes, kfDofIndexes, kfTimes, values,
    500.0f);

//-- kfDofValues Front. --//
values[+0]=  50.0f;  values[1*6 +0]= -50.0f;  values[2*6 +0]=  50.0f;
values[+1]= -45.0f;  values[1*6 +1]=  45.0f;  values[2*6 +1]= -45.0f;
values[+2]=  50.0f;  values[1*6 +2]= -50.0f;  values[2*6 +2]=  50.0f;
values[+3]= -45.0f;  values[1*6 +3]=  45.0f;  values[2*6 +3]= -45.0f;
values[+4]= -45.0f;  values[1*6 +4]=  45.0f;  values[2*6 +4]= -45.0f;
values[+5]=   0.05f; values[1*6 +5]=  -0.05f; values[2*6 +5]=   0.05f;

kfTimes[0]= 0.0f;
kfTimes[1]= 0.4f;
kfTimes[2]= 0.8f;
int flyFrontKftable = vaxActclassKftableCreate(ac, numKfDofs, numKfTimes, kfDofIndexes, kfTimes, values,
    500.0f);

//-- kfDofValues Wait. --//
values[+0]=  25.0f; values[1*6 +0]= -25.0f; values[2*6 +0]=  25.0f;
values[+1]= -45.0f; values[1*6 +1]=  45.0f; values[2*6 +1]= -45.0f;
values[+2]=  25.0f; values[1*6 +2]= -25.0f; values[2*6 +2]=  25.0f;
values[+3]= -45.0f; values[1*6 +3]=  45.0f; values[2*6 +3]= -45.0f;
values[+4]= -45.0f; values[1*6 +4]=  45.0f; values[2*6 +4]= -45.0f;
values[+5]=   0.1f; values[1*6 +5]=  -0.1f; values[2*6 +5]=   0.1f;
kfTimes[0]= 0.0f;
kfTimes[1]= 0.5f;
kfTimes[2]= 1.0f;
int flyWaitKftable  = vaxActclassKftableCreate(ac, numKfDofs, numKfTimes,kfDofIndexes, kfTimes, values,
    500.0f);
```

```
  vaSgFree(kfDofIndexes);
  vaSgFree(kfTimes);
  vaSgFree(values);

  //--- Adicion del extension slot de fly  ------------------------------//
  vaExtension *flyExt = vaxFlyInit();
  vaActclassAddExtension(ac, flyExt, -1);

  vaxActclassFlySetKftableIndexes(ac,  flyUpKftable,   flyDownKftable,   flyFrontKftable,flyWaitKftable);
  vaxActclassFlySetLodDistance(ac, 500.0f);

  //-distancia a la cual ya considera que ha alcanzado su objetivo.
  vaxActclassFlySetTargetThreshold(ac, 1.0f);

  //--- Adicion de los datos para la expresion y el speech.--------------//
  birdAcUdata *bcUdata       = (birdAcUdata *)vaActclassGetBehaviourData(ac);
  bcUdata->poseHappy       = poseExprHappy;
  bcUdata->poseUnhappy   = poseExprUnhappy;
  bcUdata->poseNeutral      = poseExprNeutral;

  bcUdata->poseBeakOpen  = poseBeakOpen;
  bcUdata->poseBeakClose = poseBeakClose;
}
```

El primer paso consiste en la obtención de los índices de los puntos de articulación y los grados de libertad mediante el uso de las funciones *vaActclassFindSklIndexByName()* y *vaActclassFindDofIndexByName()*. Mediante la función *vaxActclassLookSetLodDistances()* se indican las distancias a las cuales el mecanismo de gestión de mirada ha de actuar internamente sobre la precisión y rapidez del algoritmo usado.

A continuación se añade la extensión de control de la mirada, para ello se comienza creando una estructura de tipo *vaExtension* mediante *vaxLookInit()*, y se continúa añadiendo dicha extensión al actor mediante la función *vaActclassAddExtension()*, a la que se le pasa como último valor -1 para indicar que la coloque en el primer *extension slot* que encuentre libre. A continuación se obtienen los topes de los grados de libertad de los ojos y la cabeza mediante la función *vaActclassGetDofBounds(),* y se proporciona dicha información a la extensión de gestión de la mirada mediante la función *vaxActclassLookSetDofsLimits()*, de igual modo se le proporcionan los índices de los puntos de articulación y los grados de libertad mediante las funciones *vaxActclassLookSetSklIndexes()* y *vaxActclassLookSetDofIndexes()*.

La extensión *vaxPose* que permite fijar una parte del actor o el actor en su totalidad en una postura determinada, será empleada en la extensión *vaxBlink* definida posteriormente y también por los mecanismos internos de control de la locución y expresión. De forma similar a la anterior, se inicializa mediante la función *vaxPoseInit(),* y se asigna a la clase de actor en el primer slot libre mediante la función *vaActclassAddExtension()*. Lo que sigue son varias secuencias de creación de la tabla de índices de *dofs* y de valores de esos *dofs* y la creación de una postura que las almacene mediante la función *vaxActclassPoseCreate(),* se puede observar que dicha función retorna un valor entero que actúa como

identificador de esa postura. Se crean las tres posturas que se emplearan para el control de la expresión (*poseExprHappy*, *poseExprUnhappy* y *poseExprNeutral*), las dos que serán necesarias para el control del parpadeo (*poseOpenEyes* y *poseCloseEyes*), y las dos necesarias para el mecanismo de control de la locución (*poseBeakClose* y *poseBeakOpen*).

La extensión *vaxBlink* se inicializa de forma similar a las anteriores y se configura indicando los índices de las posturas necesarias mediante la función *vaxActclassBlinkSetPoseIndexes(),* y la distancia de desactivación del mecanismo mediante *vaxActclassBlinkSetLodDistance()*.

Las extensiones de gestión del *keyframing* y de control del vuelo son inicializadas de forma similar a las anteriores mediante las funciones *vaxKftableInit()* y *vaxFlyInit().* En este ejemplo el único objetivo de la extensión de *keyframing* es ayudar en la implementación del mecanismo de vuelo, para ello es necesaria la definición de cuatro tablas de *keyframing*, cada una de las cuales será identificada mediante un entero (*flyUpKftable*, *flyDownKftable*, *flyFrontKftable* y *flyWaitKftable*). Cada una de estas tablas de crea de un modo similar rellenando un array de *floats* con los instantes temporales en el que se define cada tabla (*kfTimes*), una array con enteros conteniendo los índices de los grados de libertad implicados (*kfDofIndexes*), y por último, un array bidimensional con los valores que adoptan los grados de libertad en cada instante de tiempo (*values*). Una vez dichos arrays han sido rellenados basta con llamar a la función *vaxActclassKftableCreate()*, la cual retorna el identificador de la tabla recién creada. Esos índices son pasados a la función *vaxActclassFlySetKftableIndexes()*. La definición de extensión de vuelo finaliza indicando la distancia de desactivación del mecanismo mediante la función *vaxActclassFlySetLODDistance(),* y también el umbral de distancia que indica cuando un actor ha alcanzado su objetivo, mediante la función *vaxActclassFlySetTargetThreshold()*.

La función *birdActclassCreateExtensions()* finaliza almacenando en las variables *poseHappy*, *poseUnhappy*, *poseNeutral*, *poseBeakOpen* y *poseBeakClose* de la estructura que almacena los datos de usuario de la vaActclass los índices de las correspondientes posturas.

La rutina ***birdVaCreatePrototypeActor()*** mostrada en el siguiente bloque define los nodos de gestión de nivel de detalle y de geometría de un actor que actuará como prototipo para la definición del resto de los actores de la clase "*Bird*", cada una de las geometrías implicadas en la definición del actor se encuentra en su propio fichero dentro del directorio indicado por el parámetro variable *geometryPath*.

```
static vaActor *birdVaCreatePrototypeActor(vaActclass *ac, char *geometryPath){
  //-- Obtencion de los indices de los nodos Skeleton ----------//
  int headIndex      = vaActclassFindSklIndexByName(ac, "head");
  int lEyeIndex      = vaActclassFindSklIndexByName(ac, "eyeL");
  int rEyeIndex      = vaActclassFindSklIndexByName(ac, "eyeR");
  int ulEyelidIndex  = vaActclassFindSklIndexByName(ac, "eyelidUL");
  int blEyelidIndex  = vaActclassFindSklIndexByName(ac, "eyelidDL");
  int urEyelidIndex  = vaActclassFindSklIndexByName(ac, "eyelidUR");
  int brEyelidIndex  = vaActclassFindSklIndexByName(ac, "eyelidDR");
  int bBeakIndex     = vaActclassFindSklIndexByName(ac, "beak");
  int tailIndex      = vaActclassFindSklIndexByName(ac, "tail");
```

```
int lWingIndex      = vaActclassFindSklIndexByName(ac, "wingL");
int rWingIndex      = vaActclassFindSklIndexByName(ac, "wingR");
int lWingtipIndex  = vaActclassFindSklIndexByName(ac, "wingtipL");
int rWingtipIndex  = vaActclassFindSklIndexByName(ac, "wingtipR");

//---Carga todos los ficheros con la geometria del actor.---------//
void *geoBodyA              = loadGeometryFromFile(geometryPath, "geoBodyA.obj");
void *geoBodyB              = loadGeometryFromFile(geometryPath, "geoBodyB.obj");
void *geoBodyC              = loadGeometryFromFile(geometryPath, "geoBodyC.obj");
void *geoBodyD              = loadGeometryFromFile(geometryPath, "geoBodyD.obj");

void *geoWingLA             = loadGeometryFromFile(geometryPath, "geoWingLA.obj");
void *geoWingLB             = loadGeometryFromFile(geometryPath, "geoWingLB.obj");
void *geoWingLC             = loadGeometryFromFile(geometryPath, "geoWingLC.obj");

void *geoWingRA             = loadGeometryFromFile(geometryPath, "geoWingRA.obj");
void *geoWingRB             = loadGeometryFromFile(geometryPath, "geoWingRB.obj");
void *geoWingRC             = loadGeometryFromFile(geometryPath, "geoWingRC.obj");

void *geoWingtipLA          = loadGeometryFromFile(geometryPath, "geoWingtipLA.obj");
void *geoWingtipLB          = loadGeometryFromFile(geometryPath, "geoWingtipLB.obj");

void *geoWingtipRA          = loadGeometryFromFile(geometryPath, "geoWingtipRA.obj");
void *geoWingtipRB          = loadGeometryFromFile(geometryPath, "geoWingtipRB.obj");

void *geoTailA              = loadGeometryFromFile(geometryPath, "geoTailA.obj");
void *geoTailB              = loadGeometryFromFile(geometryPath, "geoTailB.obj");

void *geoHeadA              = loadGeometryFromFile(geometryPath, "geoHeadA.obj");
void *geoHeadB              = loadGeometryFromFile(geometryPath, "geoHeadB.obj");
void *geoHeadC              = loadGeometryFromFile(geometryPath, "geoHeadC.obj");

void *geoEyeLA              = loadGeometryFromFile(geometryPath, "geoEyeLA.obj");
void *geoEyeLB              = loadGeometryFromFile(geometryPath, "geoEyeLB.obj");

void *geoEyeRA              = loadGeometryFromFile(geometryPath, "geoEyeRA.obj");
void *geoEyeRB              = loadGeometryFromFile(geometryPath, "geoEyeRB.obj");

void *geoEyelidULA          = loadGeometryFromFile(geometryPath, "geoEyelidULA.obj");
void *geoEyelidULB          = loadGeometryFromFile(geometryPath, "geoEyelidULB.obj");;

void *geoEyelidURA          = loadGeometryFromFile(geometryPath, "geoEyelidURA.obj");
void *geoEyelidURB          = loadGeometryFromFile(geometryPath, "geoEyelidURB.obj");

void *geoEyelidDLA          = loadGeometryFromFile(geometryPath, "geoEyelidDLA.obj");
void *geoEyelidDLB          = loadGeometryFromFile(geometryPath, "geoEyelidDLB.obj");

void *geoEyelidDRA          = loadGeometryFromFile(geometryPath, "geoEyelidDRA.obj");
void *geoEyelidDRB          = loadGeometryFromFile(geometryPath, "geoEyelidDRB.obj");

void *geoBeakA              = loadGeometryFromFile(geometryPath, "geoBeakA.obj");
void *geoBeakB              = loadGeometryFromFile(geometryPath, "geoBeakB.obj");
void *geoBeakC              = loadGeometryFromFile(geometryPath, "geoBeakC.obj");

//-- Creacion del actor propotipo.      ------------------------------------//
vaActor *actTest = vaActorNew(ac, "firstBird" );

//-- Creacion de los niveles de detalle   ------------------------------//
vaLod *lodBody = vaLodNew("lodBody", actTest);

float d1 =    5.0f;
```

```
float d2 =   20.0f;
float d3 =  100.0f;
float d4 =  500.0f;
float d5 = 2500.0f;

vaLodAddChild(lodBody, geoBodyA,              d2);
vaLodAddChild(lodBody, geoBodyB,              d3);
vaLodAddChild(lodBody, geoBodyC,              d4);
vaLodAddChild(lodBody, geoBodyD,              d5);

vaLod *lodWingL = vaLodNew("lodWingL",      actTest);
vaLodAddChild(lodWingL, geoWingLA,          d2);
vaLodAddChild(lodWingL, geoWingLB,          d3);
vaLodAddChild(lodWingL, geoWingLC,          d4);

vaLod *lodWingR = vaLodNew("lodWingR",     actTest);
vaLodAddChild(lodWingR, geoWingRA,          d2);
vaLodAddChild(lodWingR, geoWingRB,          d3);
vaLodAddChild(lodWingR, geoWingRC,          d4);

vaLod *lodWingtipL = vaLodNew("lodWingtipL", actTest);
vaLodAddChild(lodWingtipL, geoWingtipLA,     d2);
vaLodAddChild(lodWingtipL, geoWingtipLB,     d3);

vaLod *lodWingtipR = vaLodNew("lodWingtipR", actTest);
vaLodAddChild(lodWingtipR, geoWingtipRA,  d2);
vaLodAddChild(lodWingtipR, geoWingtipRB,  d3);

vaLod *lodTail = vaLodNew("lodTail",              actTest);
vaLodAddChild(lodTail, geoTailA,                  d2);
vaLodAddChild(lodTail, geoTailB,                  d3);

vaLod *lodHead = vaLodNew("lodHead",         actTest);
vaLodAddChild(lodHead, geoHeadA,             d2);
vaLodAddChild(lodHead, geoHeadB,             d3);
vaLodAddChild(lodHead, geoHeadC,             d4);

vaLod *lodEyeL = vaLodNew("lodEyeL",        actTest);
vaLodAddChild(lodEyeL, geoEyeLA,            d1 );
vaLodAddChild(lodEyeL, geoEyeLB,            d2 );

vaLod *lodEyeR = vaLodNew("lodEyeR",        actTest);
vaLodAddChild(lodEyeR, geoEyeRA,            d1 );
vaLodAddChild(lodEyeR, geoEyeRB,            d2 );

vaLod *lodEyelidUL = vaLodNew("lodEyelidUL", actTest);
vaLodAddChild(lodEyelidUL, geoEyelidULA,   d1 );
vaLodAddChild(lodEyelidUL, geoEyelidULB,   d2 );

vaLod *lodEyelidUR = vaLodNew("lodEyelidUR", actTest);
vaLodAddChild(lodEyelidUR, geoEyelidURA,   d1 );
vaLodAddChild(lodEyelidUR, geoEyelidURB,   d2 );

vaLod *lodEyelidBL = vaLodNew("lodEyelidBL", actTest);
vaLodAddChild(lodEyelidBL, geoEyelidDLA,    d1 );
vaLodAddChild(lodEyelidBL, geoEyelidDLB,   d2 );

vaLod *lodEyelidBR = vaLodNew("lodEyelidBR", actTest);
vaLodAddChild(lodEyelidBR, geoEyelidDRA,  d1 );
vaLodAddChild(lodEyelidBR, geoEyelidDRB,  d2 );
```

```
    vaLod *lodBeak = vaLodNew("lodBeak",        actTest);
    vaLodAddChild(lodBeak, geoBeakA,            d1 );
    vaLodAddChild(lodBeak, geoBeakB,            d2 );
    vaLodAddChild(lodBeak, geoBeakC,            d3 );

    vaActorAddChildSgNode(   actTest,                       vaLodGetSgNode(lodBody) );
    vaActorSklAddChildSgNode(actTest, headIndex,            vaLodGetSgNode(lodHead));
    vaActorSklAddChildSgNode(actTest, lEyeIndex,            vaLodGetSgNode(lodEyeL));
    vaActorSklAddChildSgNode(actTest, rEyeIndex,            vaLodGetSgNode(lodEyeR));
    vaActorSklAddChildSgNode(actTest, ulEyelidIndex,        vaLodGetSgNode(lodEyelidUL));
    vaActorSklAddChildSgNode(actTest, urEyelidIndex,        vaLodGetSgNode(lodEyelidUR));
    vaActorSklAddChildSgNode(actTest, blEyelidIndex,        vaLodGetSgNode(lodEyelidBL));
    vaActorSklAddChildSgNode(actTest, brEyelidIndex,        vaLodGetSgNode(lodEyelidBR));
    vaActorSklAddChildSgNode(actTest, bBeakIndex,           vaLodGetSgNode(lodBeak));
    vaActorSklAddChildSgNode(actTest, tailIndex,            vaLodGetSgNode(lodTail));
    vaActorSklAddChildSgNode(actTest, lWingIndex,           vaLodGetSgNode(lodWingL));
    vaActorSklAddChildSgNode(actTest, rWingIndex,           vaLodGetSgNode(lodWingR));
    vaActorSklAddChildSgNode(actTest, lWingtipIndex,        vaLodGetSgNode(lodWingtipL));
    vaActorSklAddChildSgNode(actTest, rWingtipIndex,        vaLodGetSgNode(lodWingtipR));
    return actTest;
}
```

La función comienza obteniendo los índices de todos los puntos de articulación del actor mediante el uso de la función *vaActclassFindSklIndexByName()*, y continúa creando los nodos con información geométrica mediante varias llamadas a la función *loadGeometryFromFile()*. La estructura básica del actor prototipo es creada mediante el uso de la función *vaActorNew()*. A continuación se realizan varias secuencias de creación de nodo de tipo *vaLod* mediante la función *vaLodNew()* y se asignan los hijos de estos nodos mediante el uso de la función *vaLodAddChild()*. Finalmente todos los nodos *vaLod* creados son enganchados en el grafo de escena del actor virtual mediante la utilización de las funciones *vaActorAddChildSgNode()* y *vaActorSklAddChildSgNode()*.

La función ***birdVaActclassCreate()*** mostrada en el siguiente bloque realiza la creación completa de la clase "*Bird*", mediante la llamada a las funciones *birdActclassCreateBasis()*, *birdActclassCreateExtension()* y *birdVaCreatePrototypeActor()* mostradas anteriormente.

```
vaActclass * birdVaActclassCreate(char *geometryPath){
    vaActclass *ac = birdActclassCreateBasis(geometryPath);
    birdActclassCreateExtensions(ac);

    vaActor *act = birdVaCreatePrototypeActor(ac, geometryPath);
    birdAcUdata *acData = (birdAcUdata *)vaActclassGetBehaviourData(ac);
    acData->protBird = act;

    BirdActclass =  ac;
    return ac;
}
```

El último fragmento del fichero *"birdVa.c"* es el mostrado a continuación, en él se puede observar la implementación de las distintas funciones que completan el "*microAPI*" de gestión de los actores de tipo "*Bird*":

```
void birdVaSetPerFrameUpdateDofsFunc(vaActclass *ac, birdBehaFunc func){
   birdAcUdata *acData = (birdAcUdata *)vaActclassGetBehaviourData(ac);
   acData->behaFunc = func;
}

void birdVaSetPerFrameUpdateRefpointFunc(vaActclass *ac, birdBehaFunc func){
   vaActclassSetRefpointBehaviourFunc( ac, func);
}

void birdVaSetSecondLookTarget(vaActor *act, float3 targetPos){
  birdUdata  *b  = (birdUdata   *)vaActorGetBehaviourData(act);
  float3Copy(b->lookSecondTarget, targetPos);
}

void birdVaSetActiveLookTarget(   vaActor *act, int activeTarget){
  birdUdata  *b  = (birdUdata   *)vaActorGetBehaviourData(act);
  b->lookActiveTarget = activeTarget;
}

void birdVaSetHappiness(vaActor *act, int happiness){ //-1, 0 o 1.
  birdUdata  *b  = (birdUdata   *)vaActorGetBehaviourData(act);
  b->newExprState = happiness;
}

void birdVaSetCurPosHpr(vaActor *act, float3 pos, float3 hpr){
    vaxActorFlySetCurPosHpr(act, pos, hpr);
}

void birdVaFlyUpdateRefpoint(vaActor *act){
   vaxActorFlyUpdateRefpoint(act);
}

void birdVaDoCheep(vaActor *act){
   birdUdata  *b  = (birdUdata   *)vaActorGetBehaviourData(act);
   b->doCheepDirty = 1;
}

void birdVaSetBlinkAttention(vaActor *act, float attention){
   vaxActorBlinkSetAttention(act, attention);
}

void birdVaSetFlyTarget(  vaActor *act,  float3 targetPos){
   vaxActorFlySetTarget(      act,  targetPos);
   vaxActorLookSetFirstTarget(act,  targetPos);
}

int birdVaIsWaiting(vaActor *act){
   return ( vaxActorFlyGetTimeWaiting(act) > 10.0f );
}

int birdVaGetFlyState(vaActor *act){
   return ( vaxActorFlyGetState(act) );
}

vaActor *birdVaCreate(float3 pos, float3 hpr, float normSpeed){
   birdAcUdata *bc = (birdAcUdata *)vaActclassGetBehaviourData( BirdActclass );
   bc->curActorNormSpeed = normSpeed;

   vaActor *act = bc->protBird;
   vaActor *actCloned = vaActorClone(act,  NULL);
   vaxActorFlySetCurPosHpr(actCloned, pos, hpr);
```

```
    return actCloned;
}

vaActclass *birdVaGetActclass(){
   return BirdActclass;
}
```

9.6 Integración de los actores virtuales en la aplicación final.

Todo el desarrollo del actor virtual ejemplo y sus módulos de extensión ha sido realizado de forma totalmente independiente de la aplicación final en la que sean integrados y del grafo de escena de escena empleado. Esto posibilita que tanto el actor como sus módulos de extensión puedan ser fácilmente reutilizados en cualquier aplicación futura.

En este apartado se describe un módulo encargado de controlar de forma global a los actores existentes en la escena. Ese módulo es el descrito como "*Actor Management Layer*" en el diagrama de arquitectura propuesto en el caplitulo anterior, y actua sobre los "*microAPI*" de los actores virtuales para conseguir que se comporten de la forma deseada.

El módulo que compone la "*Actor Management Layer*" ha de estar implementado de forma independiente de la aplicación final, y ha de presentar una serie de funciones que actúen como interface entre la aplicación de simulación y los actores virtuales.

En los siguientes subapartados se presenta la forma en la que ha sido desarrollado este módulo, la forma en la que se relaciona con la aplicación de simulación final, y por último se describe el contenido del módulo "*vaInPf*" encargado de configurar a la librería de actores para su utilización con el grafo de escena de *OpenGl Performer*.

9.6.1 Módulo de gestión de actores.

Para mejorar la modularidad de la aplicación final, se ha separado el código que gestiona la aplicación de simulación global, del código que gestiona a los actores virtuales. Esto demuestra la capacidad de programar toda la parte de la gestión de actores virtuales de un modo independiente de la aplicación final y su grafo de escena, y presenta como ventaja adicional el hecho de que permite mostrar todos los aspectos centrales de este trabajo, sin tener la necesidad de mostrar el código total de la aplicación de simulación (en este caso la aplicación "*perfly* " creada por el equipo desarrollador de *OpenGl Performer*).

La "*Actor Management Layer*" está implementada en un par de ficheros ("*birdsApp.c*" y "*birdsApp.h*"). Toda la relación entre la aplicación principal y el módulo de gestión de actores se reduce a dos funciones: la función ***sampleAppInit()*** que inicializa el módulo de gestión actores y la función ***sampleAppPerFrame()***, que actualiza todo lo relacionado con los actores virtuales en cada frame. En este ejemplo todos los actores virtuales se mueven libremente por el espacio, dependiendo directamente del sistema de coordenadas global de la escena, por esa razón no se necesita tener conocimiento del grafo de escena que representa a la ciudad, siendo suficiente con disponer de un puntero al nodo padre de toda la escena para poder conectar los nodos *Actor*. Este puntero es pasado como parámetro a la función *sampleAppInit()***.**

Respecto a la función ***sampleAppPerFrame()*** que ha de ser llamada a cada frame, se le ha de indicar la posición de cámara, un factor modulador del Nivel de detalle, y también las características del frustum, de igual modo se pasa un identificador de un evento que indica que tipo de actuación se quiere realizar sobre los actores virtuales.

El contenido del fichero *"birdsApp.h"* es el siguiente:

```
#ifndef __BIRDSAPP_H__
#define __BIRDSAPP_H__

#define EV_NONE                                 0
#define EV_CLONE_ACTOR                          1
#define EV_CLONE_MULTACTORS                     2
#define EV_DELETE_ACTOR                         3
#define EV_DELETE_MULTACTORS                    4
#define EV_REFPOINTS_FROZEN                     5
#define EV_REFPOINTS_UNFROZEN                   6
#define EV_LOOKEYE_ON                           7
#define EV_LOOKEYE_OFF                          8
#define EV_DO_CHEEP                             9

#define EV_TESTCULL_ON                          10
#define EV_TESTCULL_OFF                         11
#define EV_ACTOR_BSPHERES_SHOW                  12
#define EV_ACTOR_BSPHERES_HIDE                  13

#define EV_STATS_DISABLE_NONE                   100
#define EV_STATS_DISABLE_REFPOINTBEHAVIOUR      101
#define EV_STATS_DISABLE_SKLSROOTBEHAVIOUR      102
#define EV_STATS_DISABLE_DOFSBEHAVIOUR          103
#define EV_STATS_DISABLE_FULLBEHAVIOUR          104
#define EV_STATS_DISABLE_ACTORTRAVERSE          105
#define EV_STATS_DISABLE_MATRIXAPPLY            106
#define EV_STATS_DISABLE_DRAWING                107

extern void sampleAppInit(void *sgRootNode);
extern void sampleAppPerFrame(      int event, float3 eyePos, float lodScale,
                                    float fNear, float fFar,  float fLeft, float fRight, float fBottom, float fTop);
#endif
```

Los eventos EV_CLONE_ACTOR y EV_CLONE_MULTACTORS se emplean para generar uno o cien actores con unas características de posición y carácter al azar. Los eventos EV_DELETE_ACTOR y EV_DELETE_MULTACTORS funcionan en sentido inverso, eliminando de la escena uno o 100 actores. El evento EV_REFPOINTS_FROZEN sirven para forzar el valor de los *Reference Point* de los actores de dispongan espacialmente formando un array tridimensional, y el evento EV_REFPOINTS_UNFROZEN para quitar esa limitación. Los eventos EV_LOOKEYE_ON y EV_LOOKEYE_OFF sirven para forzar a que todos los actores miren hacia la cámara, y el evento EV_DO_CHEEP sirve para forzar a todos los actores a que simulen la emisión de un sonido.

Los eventos EV_TESTCULL_ON y EV_TESTCULL_OFF, activan y desactivan la comprobación de culling, de modo que se dibujan y procesan los actores que están fuera del frustum. Los eventos

EV_ACTOR_BSPHERES_SHOW y EV_ACTOR_BSPHERES_HIDE sirven para poder visualizar las *bounding spheres* empleadas por los actores.

Los eventos cuyo nombre comienza con EV_STATS_DISABLE, permiten desactivar algunos de los procesos relacionados con la gestión de los actores, lo cual permite evaluar de una forma práctica el coste de esos procesos: el evento EV_STATS_DISABLE_REFPOINTBEHAVIOUR desactiva la gestió de comportamiento relacionanada con el cálculo de la posición del *Reference Point* de los actores, EV_STATS_DISABLE_SKLSROOTBEHAVIOUR desactiva los cálculos relacionedos con la gestión de comportamiento de los *Skeleton Roots* de los actores, EV_STATS_DISABLE_DOFSBEHAVIOUR, desactiva la gestión de comportamiento relacionada con la actualización de los valores de los grados de libertad. EV_STATS_DISABLE_FULLBEHAVIOUR desactiva toda la gestión de comportamiento, EV_STATS_DISABLE_ACTORTRAVERSE desactiva el recorrido de los nodos que componen internamente al actor virtual, EV_STATS_DISABLE_MATRIXAPPLY desactiva la aplicación de matrices de transformación, EV_STATS_DISABLE_DRAWING desactiva el dibujado de la geometria de los actores virtuales. Por último. el evento EV_STATS_DISABLE_NONE se restauran los aspectos desactivados.

En contenido del fichero "birdsApp.c" es el siguiente:

```
#include "stdlib.h"
#include "math.h"
#include "stdio.h"

#include "vaxFly.h"
#include "birdVa.h"
#include "birdsApp.h"

//- Estructura que almacena la informacion sobre el area en la que se mueven los actores.
typedef struct sceneInfoStr{
   float3 boxCenter;
   float3 boxSize;
   float3 boxMin;
   float3 boxMax;
}sceneInfo;

static void sceneCreate(sceneInfo *scene){
  scene->boxCenter[0] = 2500.0f;
  scene->boxCenter[1] = 2500.0f;
  scene->boxCenter[2] =  100.0f;
  scene->boxSize[0]   = 3000.0f;
  scene->boxSize[1]   = 3000.0f;
  scene->boxSize[2]   =  100.0f;
 }

static float auxRandomValInMargin(float min, float max){
    float normRand = (float)rand()/RAND_MAX;
    return min + (max-min)*normRand;
}

static void sceneRandomPointInAux(sceneInfo *scene, float3 pos){
  pos[0]= auxRandomValInMargin(scene->boxCenter[0] -scene->boxSize[0]/2,                scene-
       >boxCenter[0] +scene->boxSize[0]/2);
```

```
   pos[1]= auxRandomValInMargin(scene->boxCenter[1] -scene->boxSize[1]/2,
                        scene->boxCenter[1] +scene->boxSize[1]/2);
   pos[2]= auxRandomValInMargin(scene->boxCenter[2] -scene->boxSize[2]/2,
                        scene->boxCenter[2] +scene->boxSize[2]/2);
}

static void sceneRandomPointIn(sceneInfo *scene, float3 pos){
    float3 pos1, pos2;
    sceneRandomPointInAux(scene, pos1);
    sceneRandomPointInAux(scene, pos2);
    pos[0] = (pos1[0] +pos2[0])/2.0f;
    pos[1] = (pos1[1] +pos2[1])/2.0f;
    pos[2] = (pos1[2] +pos2[2])/2.0f;
}



//-Estructura que almacena datos globales a la aplicacion.
//
typedef struct globalDataStruct{
    float3      lookTarget;         // Punto general objetivo de la mirada.
    int         activeTarget;       // Conmutador del punto objetivo de la mirada.
    void   *    sgRootNode;         // Nodo padre del grafo de escena.
    sceneInfo   scene;              // Escena en la que se mueven los actores.
    int        posFrozen;           // Flag para congelar la posicion de las actores.
    int         doCheep;            // Flag para forzar que los actore emitan un sonido.
}globalData;

globalData *vaGdata;

//-Funcion que define la posicion al que miran los actores virtuales.
//
static void defineLookTarget(globalData *data){
         float radius =  5.0f;
         float vAzim     = 80.0f;   //grados/seg;
         float angle  = vAzim* vaGetCurTime();
         if      (angle > 360.0f) angle -=  360.0f;
         else if (angle < 360.0f) angle +=  360.0f;

         float3 lookTarget;
         lookTarget[0] =  5.0f;
         lookTarget[1] =  radius*sinf(M_PI*angle/180); //vertical.
         lookTarget[2] =  radius*cosf(M_PI*angle/180);

        float3Copy(data->lookTarget, lookTarget);

         float3 eye;
         vaGetEyePosition(eye);
         float3Copy(data->lookTarget, eye);
}


//-Funcion que define los valores del refpoint de cada actor virtual a cada frame.
//
static void birdRefpointFunc(vaActor *act){
    if ( birdVaIsWaiting(act) ){
        float3 newTarget;
        sceneRandomPointIn( &(vaGdata->scene), newTarget);
        float3 actorPos;
         vaActorGetRefpointPos(act, actorPos);
        newTarget[0] = actorPos[0] +(newTarget[0] -actorPos[0])/4.0f;
```

```
        newTarget[1] = actorPos[1] +(newTarget[1] -actorPos[1])/4.0f;
        birdVaSetFlyTarget(act,         newTarget);
    }

    if ( vaGdata->posFrozen){
         vaActclass *ac = vaActorGetActclass(act);
         int numActors  = vaActclassGetNumActors(ac);
         int cubeSide   = (int)ceil(pow(numActors, 1.0f/3.0f));

        float sep = 10;

         int actIndex   = vaActorGetIndex(act);

        int segment    = actIndex/(cubeSide*cubeSide);
        int inSegment = actIndex%(cubeSide*cubeSide);
        int row           = inSegment/cubeSide;
        int col           = inSegment%cubeSide;

        float3 pos, hpr;
        pos[0] = vaGdata->scene.boxCenter[0] -cubeSide/2.0f*sep +segment*sep;
        pos[1] = vaGdata->scene.boxCenter[1] -cubeSide/2.0f*sep +row*sep;
        pos[2] = vaGdata->scene.boxCenter[2] -cubeSide/2.0f*sep +col*sep;
        hpr[0] = hpr[1] = hpr[2]= 0.0f;

        birdVaSetCurPosHpr(act, pos, hpr);
    }
    else{
        birdVaFlyUpdateRefpoint(act);
    }
}

//-Funcion que define la postura y otras propiedades del actor en cada frame.
//
static void birdBehaUpdateFunc(vaActor *act){
    birdVaSetSecondLookTarget(act, vaGdata->lookTarget);
    birdVaSetActiveLookTarget(act, vaGdata->activeTarget);
    int happiness = 0;              //neutral:0  unhappy:-1 happy:1.
    switch ( birdVaGetFlyState(act) ){
         case FLYSTATE_GOINGUP:     happiness = -1; break;
         case FLYSTATE_GOINGDOWN:
        case FLYSTATE_WAITING:        happiness =  1;  break;
        case FLYSTATE_GOINGFRONT:happiness =  0;  break;
         }
    birdVaSetHappiness(act, happiness);
    if (vaGdata->doCheep)
          birdVaDoCheep(act );
    float attention = 0.0f;
    if (vaGdata->activeTarget == 2) attention = 1.0f;
    birdVaSetBlinkAttention(act, attention);
}


//-Inicializacion de la aplicacion.
//
void sampleAppInit(void *sgRootNode){
    vaGdata                = (globalData *) vaSgMalloc(sizeof(globalData));
    globalData *gdata = vaGdata;
    vaGdata->lookTarget[0]    = 0.0f;
    vaGdata->lookTarget[1]    = 0.0f;
    vaGdata->lookTarget[2]    = 0.0f;
    vaGdata->sgRootNode     = sgRootNode;
```

```
   vaGdata->activeTarget     = 1;
   vaGdata->posFrozen        = 0;

   sceneCreate(&(vaGdata->scene));

   char *geometryPath = ".";
   vaActclass *ac  = birdVaActclassCreate(geometryPath);
   birdVaSetPerFrameUpdateDofsFunc(          ac, birdBehaUpdateFunc);
   birdVaSetPerFrameUpdateRefpointFunc(      ac, birdRefpointFunc);

   vaAddActclass( ac );
}


//-Funcion que es llamada a cada frame para actualizar los parametros
// que definen el comportamiento de la aplicacion.
// Recibe de la aplicacion principal que evento se desea ejecutar por
// ejemplo la creacion de un nuevo actor o de un grupo de ellos, y tambien
// la posicion de la camara y un valor de modulacion sobre las distancias
// a las que conmutan los niveles de detalle.
//
void sampleAppPerFrame(   int event, float3 eyePos, float lodScale,
                          float fNear, float fFar,  float fLeft, float fRight, float fBottom, float fTop){
   globalData *gdata   = vaGdata;
   vaActclass  *pclass = birdVaGetActclass();
   vaGdata->doCheep = 0;
   switch (event){
      case EV_CLONE_ACTOR:{
           float normSpeed = auxRandomValInMargin(0.0f, 1.0f);
           float3 pos;
           float3 hpr;
           hpr[0]=  hpr[1]= hpr[2] = 0.0f;
           sceneRandomPointIn( &(gdata->scene), pos);
           vaActor *actCloned =  birdVaCreate(pos, hpr, normSpeed);
           float3 target;
           sceneRandomPointIn( &(gdata->scene), target);
           vaxActorFlySetTarget(actCloned,  target);
           vaSgGrpNodeAddChild(gdata->sgRootNode, vaActorGetSgNode(actCloned) );
           printf(" EV_CLONE_ACTOR> numActors:%d\n",  vaActclassGetNumActors(pclass));
           break;
           }
      case EV_CLONE_MULTACTORS:{
           int nActors = 100;
           for (int i = 0; i<nActors; i++){
              float normSpeed = auxRandomValInMargin(0.0f, 1.0f);
              float3 pos;
              float3 hpr;
              hpr[0]=  hpr[1]= hpr[2] = 0.0f;
              sceneRandomPointIn( &(gdata->scene), pos);
              vaActor *actCloned =  birdVaCreate(pos, hpr, normSpeed);
              vaSgGrpNodeAddChild( gdata->sgRootNode,  vaActorGetSgNode(actCloned));
              float3 target;
              sceneRandomPointIn( &(gdata->scene), target);
              vaxActorFlySetTarget(actCloned,  target);
           }
           printf(" EV_CLONE_MULTACTORS> numActors:%d\n", vaActclassGetNumActors(pclass));
           break;
           }
      case EV_DELETE_ACTOR:{
           int numActors = vaActclassGetNumActors(pclass);
           if (numActors > 2){
```

```
            vaActor *act = vaActclassGetActor( pclass, numActors -1);
             vaActorDelete(act);
             vaSgGrpNodeRemoveChild(gdata->sgRootNode,vaActorGetSgNode(act));
          }
          printf(" EV_DELETE_ACTOR> numActors:%d\n",  vaActclassGetNumActors(pclass));
          break;
          }
    case EV_DELETE_MULTACTORS:{
          int nActorsToDelete = 100;
          int numActors = vaActclassGetNumActors(pclass);
          for (int i = 0; i<nActorsToDelete; i++){
            if (numActors > 2){
              vaActor *act = vaActclassGetActor( pclass, numActors -1);
                    vaActorDelete(act);
                    vaSgGrpNodeRemoveChild(gdata->sgRootNode,  vaActorGetSgNode(act));
              }
          }
          printf(" EV_DELETE_MULTACTORS> numActors:%d\n",  vaActclassGetNumActors(pclass));
          break;
          }
    case EV_REFPOINTS_FROZEN:
          gdata->posFrozen = 1;
          break;
     case EV_REFPOINTS_UNFROZEN:
          gdata->posFrozen = 0;
          break;
     case EV_LOOKEYE_ON:
          vaGdata->activeTarget = 2;
          printf("==>Lookat ActiveTarget:%d\n", vaGdata->activeTarget);
          break;
     case EV_LOOKEYE_OFF:
          vaGdata->activeTarget = 1;
          printf("==>Lookat ActiveTarget:%d\n", vaGdata->activeTarget);
          break;
    case EV_DO_CHEEP:
          vaGdata->doCheep = 1;
          printf("==>force Cheep.\n");
          break;
     case EV_TESTCULL_ON:
          vaStatsSetTestCullProcess( 1);
          printf("==>Visualizacion de actores fuera del frustum On.\n");
          break;
     case EV_TESTCULL_OFF:
          vaStatsSetTestCullProcess( 0);
          printf("==>Visualizacion de actores fuera del frustum Off.\n");
          break;
     case EV_ACTOR_BSPHERES_SHOW:
          printf("==>Activada la visualizacion de bspheres de Actores.\n");
          vaStatsSetViewBSpheres( 1);
          break;
     case EV_ACTOR_BSPHERES_HIDE:
          printf("==>Desactivada la visualizacion de bspheres de Actores.\n");
          vaStatsSetViewBSpheres( 0);
          break;
     case EV_STATS_DISABLE_NONE:
          vaStatsSetMode(       VASTATS_DISABLE_NONE, 1);
          break;
     case EV_STATS_DISABLE_REFPOINTBEHAVIOUR:
          vaStatsSetMode(       VASTATS_DISABLE_REFPOINTBEHAVIOUR, 1);
          break;
     case EV_STATS_DISABLE_SKLSROOTBEHAVIOUR:
```

```
            vaStatsSetMode(      VASTATS_DISABLE_SKLSROOTBEHAVIOUR, 1);
            break;
        case EV_STATS_DISABLE_DOFSBEHAVIOUR:
            vaStatsSetMode(      VASTATS_DISABLE_DOFSBEHAVIOUR,    1);
            break;
        case EV_STATS_DISABLE_ACTORTRAVERSE:
            vaStatsSetMode(      VASTATS_DISABLE_ACTORTRAVERSE,      1);
            break;
        case EV_STATS_DISABLE_MATRIXAPPLY:
            vaStatsSetMode(      VASTATS_DISABLE_MATRIXAPPLY,          1);
            break;
        case EV_STATS_DISABLE_DRAWING:
            vaStatsSetMode(      VASTATS_DISABLE_DRAWING,        1);
            break;
        default:
            break;
    }
    defineLookTarget(gdata); //fija el punto al que miran los actores.

    vaSetEyePosition( eyePos ); //Indica a la libreria de gestion de actores la posicion actual de la camara.
    vaSetLodScale(lodScale); //Indica a la libreria de gestion de actores el factor de modulacion para los LODs.

    static int counter = 0;
    counter ++;
    if (counter == 50){
        vaStatsShow();
        counter = 0;
        }

    vaUpdateSim();  //Funcion principal de actualizacion del estado   de todos los actores virtuales.
}
```

Se puede observar como el fichero emplea dos estructuras de datos auxiliares: ***sceneInfo,*** que se emplea para describir las características de un paralelepípedo en el interior del cual volaran los actores, y la estructura ***globalData*** que almacena algunos campos necesarios para la simulación.

Las funciones auxiliares ***sceneCreate()*** y ***sceneRandomPointIn()*** respectivamente definen las dimensiones de la escena, y generan un punto al azar dentro de ese volumen. La función ***defineLookTarget()*** se emplea para especificar el objetivo secundario de mirada de los actores, en este momento este punto está siendo la posición actual de la cámara, pero, además, se ha añadido el código que definiría un punto describiendo una trayectoria circular.

La función ***birdRefpointFunc()*** es usada como rutina de *callback* para actualizar la posición de los *Reference Point* de los actores a cada frame. Se puede observar como consulta si un actor está parado, asignándole en tal situación otras nuevas coordenadas a las que ir. En el caso de que se quiera visualizar los actores parados, se bloquea la llamada a la actualización del mecanismo de vuelo y se fijan directamente las posiciones de cada actor mediante la función *birdVaSetCurPosHpr()*.

En la función ***birdBehaUpdateFunc()*** se actualiza el segundo punto de mirada para cada uno de los actores virtuales, y también se actúa sobre la expresión facial haciendo que se muestren felices si están descendiendo o esperando, e infelices si tienen que desplazarse hacia arriba. También se fuerza a que emitan un sonido en caso de que sea necesario, o a que modifiquen su forma de parpadeo en el caso de que estén mirando al objetivo de mirada secundario.

9.6.2 Integración del módulo de gestión de actores en la aplicación de simulación global.

El ejemplo de utilización de actores virtuales ha sido integrado dentro de la aplicación de exploración de databases de *OpenGl Performer* conocida como "*perfly*". En la implementación de dicha aplicación se ven implicados varios ficheros de cierta complejidad cuya introducción en esta memoria de tesis no resultaría adecuada. En su lugar se va a presentar una visión esquemática de su estructura interna consistente en considerar que hay una zona de código en la que se inicializa a la aplicación, otra zona en la que se finaliza, y por último, otra que es llamada a cada frame. Para más entendible la integración, se va a suponer, también, que la aplicación "*perfly*" ha sido ampliada con las funciones ***perflyGetSceneGroup()*** que retorna un puntero al nodo padre de todo el grafo de escena; ***perflyGuiGetEvent()*** que accede al sistema de gestión de eventos propio de la aplicación ampliado previamente para retornar los eventos indicados en el fichero "*birdsApp.h*"; la función ***perflyGetEyePosition()*** y ***perflyGetLodScale()*** que retornan la posición actual de la cámara y el factor de modulación de LOD empleado por la aplicación "*perfly*", y la función ***perflyGetFrustumPlanes()***, que retorna los parámetros que definen la pirámide de visión.

Con las consideraciones anteriores, la integración de bloque de gestión de los actores virtuales en la aplicación de simulación quedaría reducida a realizar las siguientes modificaciones en el fichero principal de la aplicación:

```
Ficheros que han de ser incluidos adicionalmente:
    #include "vaApi.h"
    #include "vaInPf.h"
    #include "birdsApp.h"


Código que ha de ser agregado en la zona de inicialización:
    pfGroup *sceneGroup = perflyGetSceneGroup();
    pfFilePath("/usr/share/Performer/data/town");
    pfNode * root = pfdLoadFile("/usr/share/Performer/data/town/town_ogl_pfi.pfb");
    pfAddChild(sceneGroup,  root);

    vaInit();
    configureVaToPerfomer();
    sampleAppInit( (void*) sceneGroup );


Código que ha de ser llamado a cada frame:
     int    event       =  perflyGuiGetEvent();
```

```
        float3  eyePosition  = perflyGetEyePosition();
        float    lodScale      =  perflyGetLodScale();
         float   pNear, pFar, pLeft, pRight, pBottom, pTop;
         perflyGetFrustumPlanes( &pNear,  &pFar, & pLeft,  &pRight, &pBottom, &pTop);
        sampleAppPerFrame(event, eyePosition,  lodScale, pNear,  pFar, pLeft,  pRight, pBottom, pTop);

Código que ha de ser llamando al finalizar la aplicación:
        vaEnd();
```

9.6.3 Adaptación para el uso con el grafo de escena OpenGl Performer.

Por último, se muestra el módulo encargado de configurar a la librería de actores para su trabajo con el grafo de escena *OpenGl Performer*. Todo el código encargado de esta configuración se encuentra en el fichero *"vaInPf.c"*. Este fichero supone un buen ejemplo de utilización de las funciones contenidas en el fichero *"vaSgCfg.h"*. La función principal del fichero *"vaInPf.c"* es ***configureVaToPerfomer()***, en ella se pasa a la librería de gestión de actores los punteros a las funciones necesarias para trabajar empleando dicho grafo de escena.

Para configurar la aplicación de actores virtuales para el trabajo con *OpenGl Performer* bastaría con incluir en el proyecto el fichero *"vaInPf.c"* y su archivo de cabecera *"vaInPf.h"* y realizar en la fase de inicialización una llamada a la función ***configureVaToPerfomer()***.

El contenido del fichero "*vaInPf.h*" es el siguiente:

```
#ifndef __VA_INPF_H__
#define __VA_INPF_H__

extern void configureVaToPerfomer();

#endif
```

El contenido del fichero "*vaInPf.c*" sería:

```
#include "stdlib.h"

#define PF_C_API 1
#include <Performer/pf.h>
#include <Performer/pf/pfSwitch.h>
#include <Performer/pf/pfTraverser.h>


#include "vaApi.h"
#include "vaSgcfg.h"

//---- funciones de gestion de Memoria ---------------------------------------//

void *sgMallocFunc(size_t nbytes){
   return pfMalloc(nbytes, pfGetSharedArena() );
   }
void *sgReallocFunc(void *ptr, size_t nbytes){
```

```
   return pfRealloc(ptr, nbytes );
   }
void sgFreeFunc(void *ptr){
   pfFree(ptr );
   }

//---- Funciones de gestion de Nodos de tipo Grupo ----------------------------//

void *sgGroupCreate(char *name){
   pfNode *grp= (pfNode *) (pfNewGroup());
   pfNodeName(grp, name);
   return (void*)grp;
   }
void sgGroupDelete(void *grpNode){
   pfDelete((pfNode*) grpNode);
   }
void sgGroupAddChild(void *parent, void *child){
   pfAddChild( (pfGroup*)parent, (pfNode*)child);
   }
void  sgGroupInsertChild(void *parent, int index, void *child){
   pfInsertChild( (pfGroup*)parent, index, (pfNode*)child);
   }
void sgGroupRemoveChild(void *parent, void *child){
   pfRemoveChild((pfGroup*)parent, (pfNode*)child);
   }
int  sgGroupGetNumChildren(void *grpNode){
   return pfGetNumChildren( (pfGroup*)grpNode);
   }
void *sgGroupGetChild(void *grpNode, int nChild){
   return (void*)pfGetChild( (pfGroup*)grpNode, nChild) ;
   }

//---- Funciones de gestion de Nodos de tipo Switch ---------------------------//

void *sgSwitchCreate(char *name){
   pfNode *swn= (pfNode *) (pfNewSwitch());
   pfNodeName(swn, name);
   return (void*)swn;
}
void sgSwitchDelete(void *swNode){
   pfDelete((pfSwitch*)swNode);
}

void sgSwitchSelectChild(void *sw, float child){
   if (child == -1.0f)
        pfSwitchVal((pfSwitch *)sw, PFSWITCH_OFF);
   else
        pfSwitchVal((pfSwitch *)sw, child);
   }

//---- Funciones de gestion de datos de Usuario dentro de un nodo ------------//

void sgNodeSetUData(void *sgNode, void *data){
   pfUserData( (pfObject*)sgNode, data);
   }
void * sgNodeGetUData(void *sgNode){
   int numUData = pfGetNumUserData( (pfObject*)sgNode );
   if (numUData == 0)
        return NULL;
   return pfGetUserData( (pfObject*)sgNode);
   }
```

```
//---- Funciones de callback asociadas a los nodos ---------------------------//

int sgpfActnodePrecullCb(pfTraverser *trav,  void *data){
   pfNode *sgNode  =   pfGetTravNode(trav);
   float16 sgInitialMat;

   pfMatrix mat ;
   pfGetTravMat( trav, mat);
   float16Copy(sgInitialMat, (float*)&(mat[0][0]));

   int precullResult = vaActNodePreCull((void *)sgNode,  sgInitialMat, (void*)trav );
   int retVal;
   switch (precullResult){
        case VA_PRECULLRESULT_CONT:
             //-Deja que Performer aplique su mecanismo de culling por defecto.
             retVal =  PFTRAV_CONT;
             break;
        case VA_PRECULLRESULT_CONT_ALLIN:
              //-Este nodo y sus hijos de este deben ser aceptados directamente sin ningun
             // tipo de comprobacion adicional.
             pfCullResult(PFIS_MAYBE|PFIS_TRUE|PFIS_ALL_IN);
             retVal =  PFTRAV_CONT;
             break;
        case VA_PRECULLRESULT_PRUNE:
             //-El nodo y sus hijos no deben ser dibujados.
             pfCullResult(PFIS_FALSE);
             retVal = PFTRAV_PRUNE;
             break;
        }
   return retVal;
}

int sgpfActnodePredrawCb(pfTraverser *trav,  void *data){
   pfNode  *sgNode = pfGetTravNode( trav );
   vaActNodePreDraw((void *)sgNode);
   return 0;
}

int sgpfActnodePostdrawCb(pfTraverser *trav,  void *data){
   pfNode  *sgNode = pfGetTravNode( trav );
   vaActNodePostDraw((void*) sgNode);
   return 0;
}

int sgpfLodnodePrecullCb(pfTraverser *trav,  void *data){
   pfSwitch *sw   = (pfSwitch *)pfGetTravNode(trav);
   vaLodNodePrecull( (void *)sw, (void*) trav);
   return(PFTRAV_CONT);
}

int sgpfSklnodePrecullCb(pfTraverser *trav,  void *data){
   pfNode *sgNode    = pfGetTravNode(trav);
   int precullResult = vaSklNodePreCull( (void*)sgNode, (void *)trav);
   if (precullResult == VA_PRECULLRESULT_PRUNE){
        pfCullResult(PFIS_FALSE);
        return (PFTRAV_PRUNE);
       }
  return PFTRAV_CONT;
}
```

```
int sgpfSklnodePredrawCb(pfTraverser *trav,  void *data){
   pfNode *sgNode  = pfGetTravNode(trav);
   vaSklNodePreDraw(sgNode);
   return 0;
}

int sgpfSklnodePostdrawCb(pfTraverser *trav,  void *data){
   pfNode *sgNode  = pfGetTravNode(trav);
   vaSklNodePostDraw((void*)sgNode);
   return 0;
}

void sgActNodeDefineTravFuncs(void *sgNode){
   pfNodeTravFuncs( (pfNode*)sgNode , PFTRAV_CULL,   (pfNodeTravFuncType ) sgpfActnodePrecullCb, NULL);
   pfNodeTravFuncs( (pfNode*)sgNode, PFTRAV_DRAW, (pfNodeTravFuncType ) sgpfActnodePredrawCb,
                                                  (pfNodeTravFuncType ) sgpfActnodePostdrawCb);

}

void  sgSklNodeDefineTravFuncs(void *sgNode){
   pfNodeTravFuncs( (pfNode *) sgNode, PFTRAV_DRAW, (pfNodeTravFuncType) sgpfSklnodePredrawCb,
                                                    (pfNodeTravFuncType) sgpfSklnodePostdrawCb);

   pfNodeTravFuncs( (pfNode *) sgNode, PFTRAV_CULL,  (pfNodeTravFuncType) sgpfSklnodePrecullCb,  NULL);
}

void   sgLodNodeDefineTravFuncs(void *sgNode){
   pfNodeTravFuncs( (pfNode *)sgNode, PFTRAV_CULL,  (pfNodeTravFuncType) sgpfLodnodePrecullCb, NULL);
   }


//--- Funciones varias ----------------------------------------------------//
float sgGetTime(){
   return pfGetTime();
   }

void sgApplyBsphereToNode(void *node, float4 sph){
   pfSphere bsph;
   bsph.radius = sph[3];
   pfSetVec3(bsph.center,     sph[0], sph[1], sph[2]);
   pfNodeBSphere((pfNode *)node, &bsph ,PFBOUND_STATIC);
   }

int sgGetCullResult(void *sgNode, void *data){
   int result = pfGetCullResult();
   int vaResult = VA_PRECULLRESULT_CONT;
   switch (result){
      case PFIS_FALSE:
         vaResult = VA_PRECULLRESULT_PRUNE;
         break;
      case PFIS_MAYBE | PFIS_TRUE:
         vaResult = VA_PRECULLRESULT_CONT;
         break;
      case PFIS_MAYBE | PFIS_TRUE | PFIS_ALL_IN:
         vaResult = VA_PRECULLRESULT_CONT_ALLIN;
         break;
      }
   return vaResult;
   }
```

```
void *sgNodeCloneTree(void *sgNode){
   pfNode * clonedTree =  pfClone((pfNode*)sgNode , 0);
   return (void*)clonedTree;
   }

void configureVaToPerfomer(){
   vaSgSetMemoryFuncs(     (vaSgMallocFunc) sgMallocFunc,
                           (vaSgReallocFunc)sgReallocFunc,
                           (vaSgFreeFunc)   sgFreeFunc);

   vaSgSetGroupNodeFuncs((vaSgGrpNodeCreateFunc)            sgGroupCreate,
                           (vaSgGrpNodeDeleteFunc)          sgGroupDelete,
                           (vaSgGrpNodeAddChildFunc)        sgGroupAddChild,
                           (vaSgGrpNodeInsertChildFunc)     sgGroupInsertChild,
                           (vaSgGrpNodeRemoveChildFunc)     sgGroupRemoveChild,
                           (vaSgGrpNodeGetNumChildrenFunc)  sgGroupGetNumChildren,
                           vaSgGrpNodeGetChildFunc)         sgGroupGetChild);

   vaSgSetSwitchNodeFuncs( (vaSgSwNodeCreateFunc)           sgSwitchCreate,
                           (vaSgSwNodeDeleteFunc)           sgSwitchDelete,
                           (vaSgSwNodeSelectChildFunc)      sgSwitchSelectChild);

   vaSgSetNodeUserDataFuncs(   (vaSgNodeSetUDataFunc)        sgNodeSetUData,
                           (vaSgNodeGetUDataFunc)           sgNodeGetUData);

   vaSgSetNodesCbFuncs( (vaSgNodeSetCbFuncsFunc)            sgActNodeDefineTravFuncs,
                            (vaSgNodeSetCbFuncsFunc)        sgSklNodeDefineTravFuncs,
                            (vaSgNodeSetCbFuncsFunc)        sgLodNodeDefineTravFuncs);

   vaSgSetGetTimeFunc(       (vaSgGetTimeFunc)              sgGetTime);
   vaSgSetApplyBsphToNodeFunc( (vaSgApplyBSphToNodeFunc) sgApplyBsphereToNode);
   vaSgSetNodeCullResultFunc(  (vaSgGetCullResultFunc)      sgGetCullResult);
   vaSgSetCloneTreeFunc(      (vaSgCloneTreeFunc)           sgNodeCloneTree);
   }
```

La forma en la que se personaliza el culling desde *OpenGl Performer* encierra cierta complejidad, y a pesar de que ha sido introducido de forma genérica en el apartado "*Gestión de las funciones de callback a los nodos*" del capítulo anterior, para una total comprensión del código es aconsejable consultar el apartado *"Controlling and Customizing Traversals"* dentro la documentación proporcionada por OpenGl Performer.

9.7 Resultados.

En los anteriores apartados se ha mostrado todos los módulos necesarios para la implementación de un ejemplo de aplicación empleando las estructuras y métodos propuestos en este trabajo. El resultado final ha sido una versión ampliada de la aplicación "perfly" de OpenGl Performer, que incorpora y permite gestionar múltiples actores virtuales. En este apartado se muestran distintos aspectos de esta aplicación final, indicando la forma de interacción entre el usuario y la aplicación, varias imagenes que muestran distintos aspectos de la aplicación en funcionamiento, y, por último, varias observaciones sobre la modularidad del proyecto total y el rendimiento de la aplicación.

9.7.1 Control de la aplicación.

El código original de la aplicación "*perfly*" ha sido modificado de la forma indicada anteriormente (apartado 9.6.2), de modo que cuando se ejecuta, aparece mostrando la base de datos gráfica conocida como "*Performer Town*". El control de la aplicación se realiza a través de la interfaz gráfica proporcionado originalmente por la aplicación "*perfly*" (mostrado de forma simplificada en la *Figura 9-9*), al que se han añadido algunos eventos nuevos referentes a la gestión de los actores virtuales. Estos eventos son generados mediante la pulsación de alguna tecla.

Figura 9-9. Interfaz de usuario básico de la aplicación perfly.

La aplicación "*perfly*" permite seleccionar varios modos de exploración de la escena, tales como caminar sobre el suelo, volar, o observarla simulando un "*trackball*". También dispone de un control que permite forzar el criterio de selección de nivel de detalle para poder observar los objetos con más o menos calidad, de un editor que permite hacer que el frustum con el que se realiza el culling sea más estrecho que el campo de visión, esto permite comprobar si la gestión del culling está siendo realizada de una forma adecuada. Existen otros controles que permiten modificar las condiciones atmosféricas, el modo en el que se dibuja (sólido, alambre, puntos), la iluminación global, etc. También proporciona una serie de controles que poder estudiar el rendimiento de la aplicación en cada instante, así como la capacidad de visualización de una representación esquemática del grafo de escena.

Respecto a interacción con gestión de los actores virtuales, se realiza totalmente mediante el teclado, generando los eventos los indicados en el fichero "*birdsApp.h*". Su vinculación con la pulsación de teclas del es la siguiente:

Tecla	***Evento / eventos***	***Acción***
a	EV_TESTCULL_ON / EV_TESTCULL_OFF	Activa/desactiva la visualización de actores fuera del *frustrum*.
b	EV_REFPOINTS_FROZEN / EV_REFPOINTS_UNFROZEN	Congela/Descongela la posición del *Reference Point* de los actores.
e	EV_CLONE_ACTOR	Crea un nuevo actor por clonación.
E	EV_DELETE_ACTOR	Elimina un actor.
f	EV_CLONE_MULTACTORS	Crea 100 nuevos actores por clonación.
F	EV_DELETE_MULTACTORS	Elimina 100 actores.
g	EV_ACTOR_BSPHERES_SHOW / EV_ACTOR_BSPHERES_HIDE	Activa/desactiva la visualización de *bounding spheres* de los actores.
c	EV_DO_CHEEP	Fuerza a que los actores abran pico.
l	EV_LOOKEYE_ON / EV_LOOKEYE_OFF	Fuerza a que los actores miren hacia la cámara.
1	EV_STATS_DISABLE_REFPOINTBEHAVIOUR	Desactiva *Refpoint Behaviour*
2	EV_STATS_DISABLE_SKLSROOTBEHAVIOUR	Desactiva *Sklsroot Behaviour*.
3	EV_STATS_DISABLE_DOFSBEHAVIOUR	Desactiva *Dofs Behaviour*.
4	EV_STATS_DISABLE_ACTORTRAVERSE	Desactiva el traverse en los nodos Actor
5	EV_STATS_DISABLE_MATRIXAPPLY	Desactiva la aplicación de matrices en nodos *Actor* y *Skeleton.*
6	EV_STATS_DISABLE_DRAWING	Desactiva el dibujado de los nodos de geometría de los actores.
0	EV_STATS_DISABLE_NONE	Restaura los aspectos desactivados.

Tabla 9-13. Vinculación entre la pulsación de teclas y las acciones sobre la simulación.

9.7.2 Aplicación en funcionamiento.

En este apartado se muestran varias imágenes relacionadas con distintos aspectos de la aplicación en funcionamiento

La escena de simulación en la que se han integrado los actores virtuales ha sido el modelo de una pequeña ciudad y su entorno circundante proporcionada como ejemplo de utilización de la librería *OpenGl Performer*. Este database, conocido como "*Performer Town*", es mostrado en la *Figura 9-10*, también aparece representado un fragmento de su grafo de escena. Los actores virtuales desarrollarán sus actividades en el cielo de dicha ciudad, y sus subgrafos de escena serán integrados en el grafo de escena global.

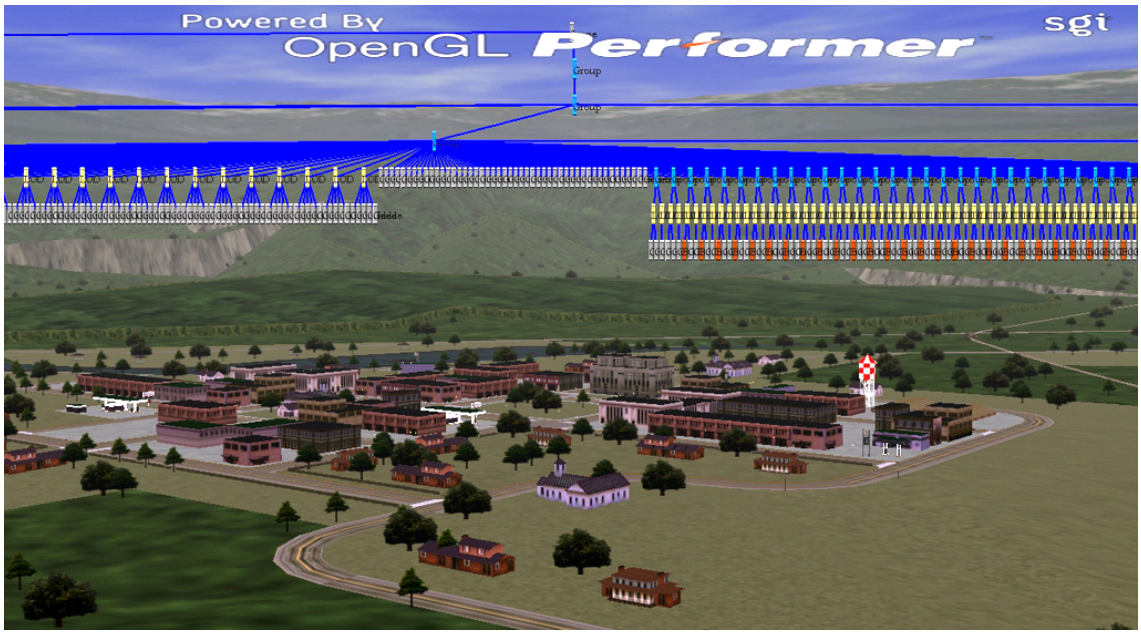


Figura 9-10. Vista de la escena "*Performer Town*" y su grafo de escena.

En la siguiente imagen (*Figura 9-11*) se muestra una representación de uno de los actores virtuales integrado en la simulación anterior, y sobredibujado aparece su grafo de escena. En esta representación se puede observar a los nodos *Actor* y *Skeleton* representados por medio de un nodo grupo de *OpenGl Performer* (*pfGroup*), y a los nodos de gestión de nivel de detalle (*vaLod*), implementados por medio de un nodos de tipo "*switch*" de *OpenGl Performer* (*pfSwitch*), también se pueden observar los nodos que almacenan la información geométrica de las distintas partes del actor (nodos *pfGeode*).

Se puede observar la correspondencia entre el grafo de escena mostrado en la figura anterior, y el presentado en la *Figura 9-7*.

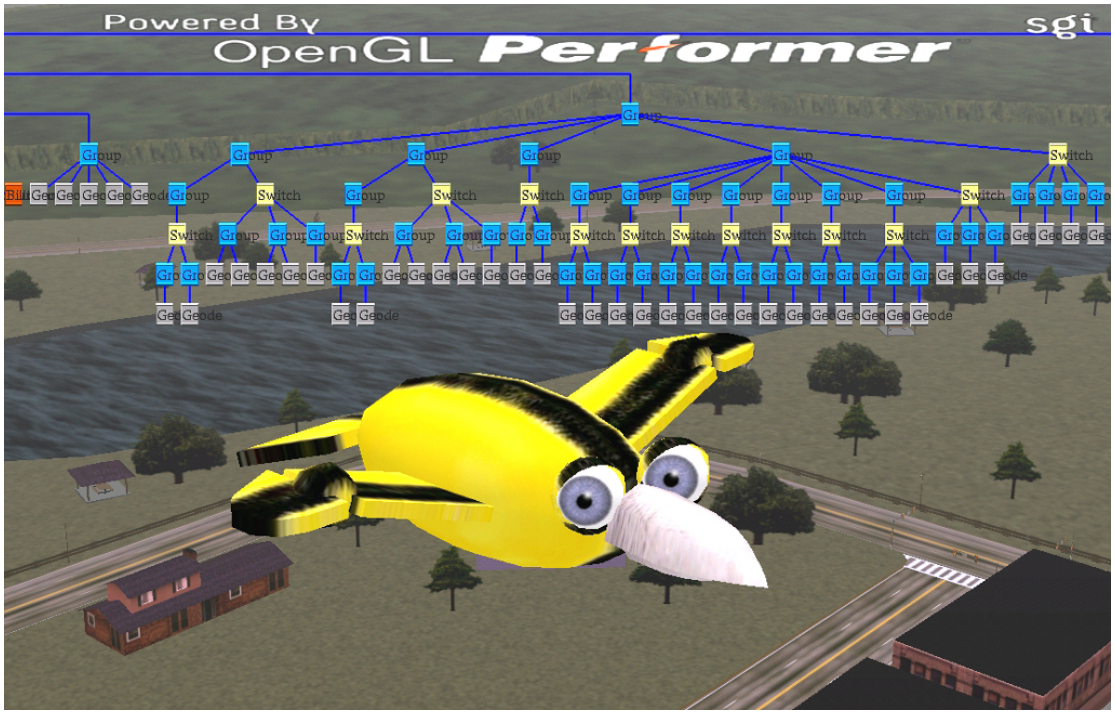


Figura 9-11. Representación del actor virtual y su grafo de escena.

En la siguiente imagen (*Figura 9-12*) se muestran los distintos tipos de geometría externa que pueden adoptar los actores de este ejemplo. Cada uno de los cuatro actores mostrados, presenta un comportamiento totalmente autónomo y su propia personalidad.

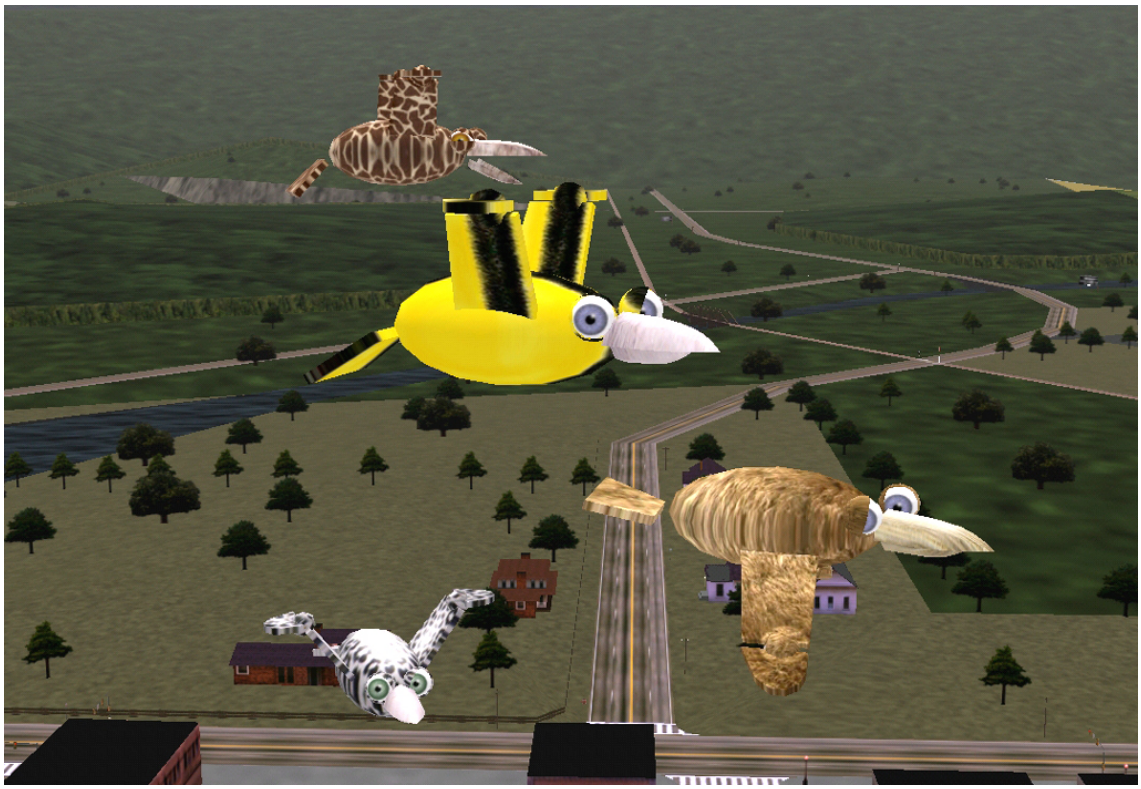


Figura 9-12. Creación de actores por medio de la clonación de un actor prototipo.

En este caso, los actores comparten los mismos nodos de geometría, ésta es la forma más sencilla de crear un actor virtual, además, presenta la ventaja de que la creación de un actor nuevo resulta muy rápida, puesto que no es necesario realizar ninguna copia de los nodos con información geométrica. Para hacer el método un poco más sofisticado, se ha añadido a este patrón básico la capacidad de que cada actor pueda tener una textura diferente, esto es conseguido mediante la utilización de rutinas de callback que son asignadas a los nodos *Actor*, y que se encargan aplicar la textura con la que serán dibujadas sus geometrías.

En la siguiente imagen (*Figura 9-13*), se puede observar una escena en la que existen varios actores virtuales situados a diferentes distancias de la cámara, y presentando, por tanto, diferentes niveles de detalle.

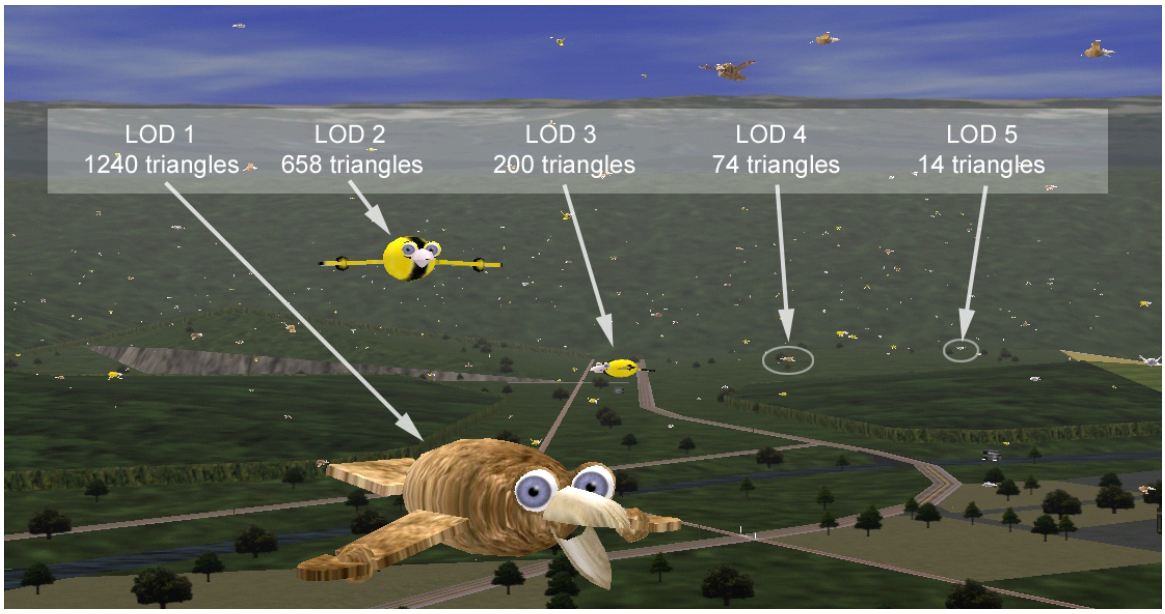


Figura 9-13. Selección automática del nivel de detalle con la distancia.

Como se puede observar, el hecho de que los actores virtuales que están lejos presenten menos polígonos, no es fácil de apreciar a simple vista: Las distancias de cambio de nivel de detalle están ajustadas para que la pérdida de número de triángulos no afecte a la calidad de la imagen. La gestión de nivel de detalle afecta también a la topología de los actores, reduciendo el número de puntos de articulación a emplear en función de la distancia, y también a su gestión de comportamiento, simplificando los cálculos, o incluso bloqueando algunos mecanismos.

En la siguiente imagen (*Figura 9-14*), se ha empleado el editor de la interface gráfica que permite modular el criterio de selección de nivel de detalle para reducir artificialmente la cantidad de triángulos empleados en la escena, de este modo se hace evidente la pérdida de calidad en la imagen, que no solamente se aprecia en los actores, sino también en los objetos de la ciudad. Esta pérdida de calidad en la imagen afecta de forma coherente al tiempo necesario para dibujar cada frame, pudiendo ser empleado para mantener tasas de actualización constantes.

Figura 9-14. Utilización del factor modulación del nivel de detalle para reducir la calidad de la imagen (y disminuir el tiempo de dibujado).

En la siguiente imagen (*Figura 9-15*), se muestra la forma en la que el culling de actores virtuales actúa de forma paralela al culling estándar del grafo de escena. Se ha empleado el editor de control de la pirámide de culling proporcionado por la aplicación "*perfly*", para hacer que la pirámide con la que se realiza el culling sea más reducida que la pirámide de visión. De este modo se puede observar si el culling está actuando de la forma adecuada. Se muestran dos imágenes, la primera de ellas contiene la escena original, y la segunda muestra la escena con un *culling* forzado. El rectángulo de la segunda imagen indica los márgenes de la pirámide de culling, se puede observar que ciertos actores han dejado de ser dibujados, de la misma forma que ha ocurrido con ciertos objetos de la "*Performer Town*".

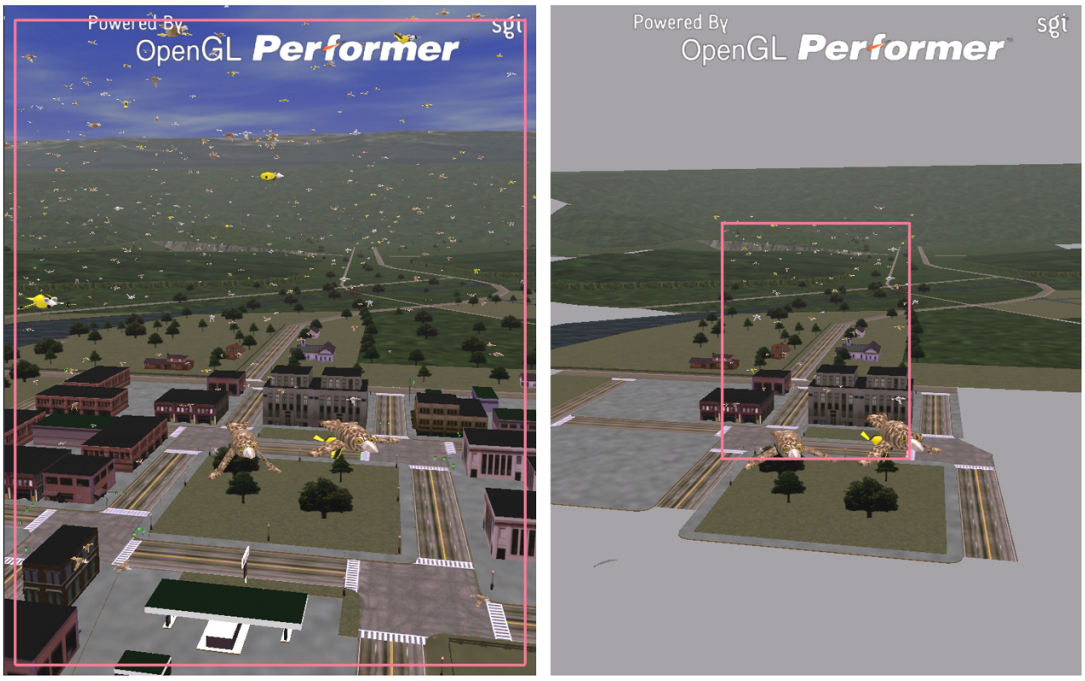


Figura 9-15. Integración entre el culling de actores y el culling del grafo de escena.

En las siguientes imágenes (*Figura 9-16*), se muestra la forma en la que actúa el mecanismo de control de mirada. En la primera imagen, los actores están mirando al punto al que se dirigen, en la segunda, los actores miran hacia la cámara. Este cambio se realiza mediante la pulsación de la tecla "l" que está vinculada con los eventos EV_LOOKEYE_ON y EV_LOOKEYE_OFF, y que finalmente acaba actuando sobre el comando *birdVaSetActiveLookTarget()* del microAPI del actor. El punto principal de mirada es asignado automáticamente dentro del código de "*birdVa.c*" al punto destino de vuelo, y el punto secundario a la posición de la cámara.

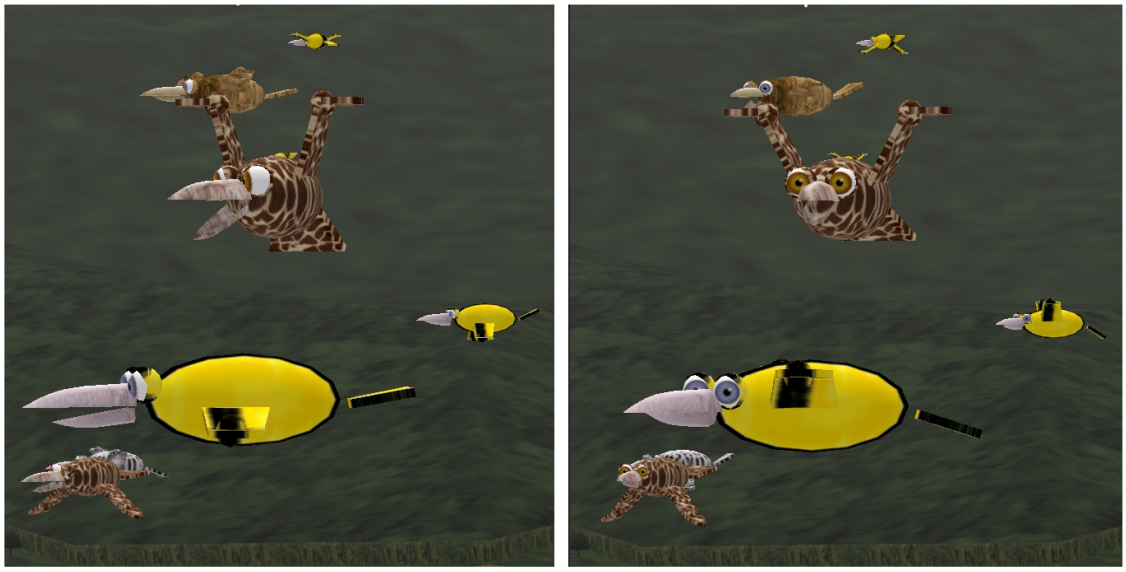


Figura 9-16. Forma de actuación del mecanismo de control de la mirada.

En la siguiente imagen (*Figura 9-17*), se muestra la forma en la que actúa el culling de actores. Para generar dicha imagen, se ha reducido la pirámide de culling mediante el control proporcionado por la utilidad "*perfly*", se ha activado la visualización de actores fuera del frustum mediante la generación de un evento EV_TESTCULL_OFF (tecla "a"), y se ha activado la visualización de las *bounding spheres* estáticas de los actores, mediante la generación de un evento EV_ACTOR_BSPHERES_SHOW (tecla "g"). En dicha figura se pueden observar tanto la *Repoint Bspere* como la *Sklsroot Bsphere* de cada uno de los actores presentes en la escena. El cambio de color existentes entre las esferas que están dentro y fuera del frustum muestra la forma en la que actúa el culling de actores.

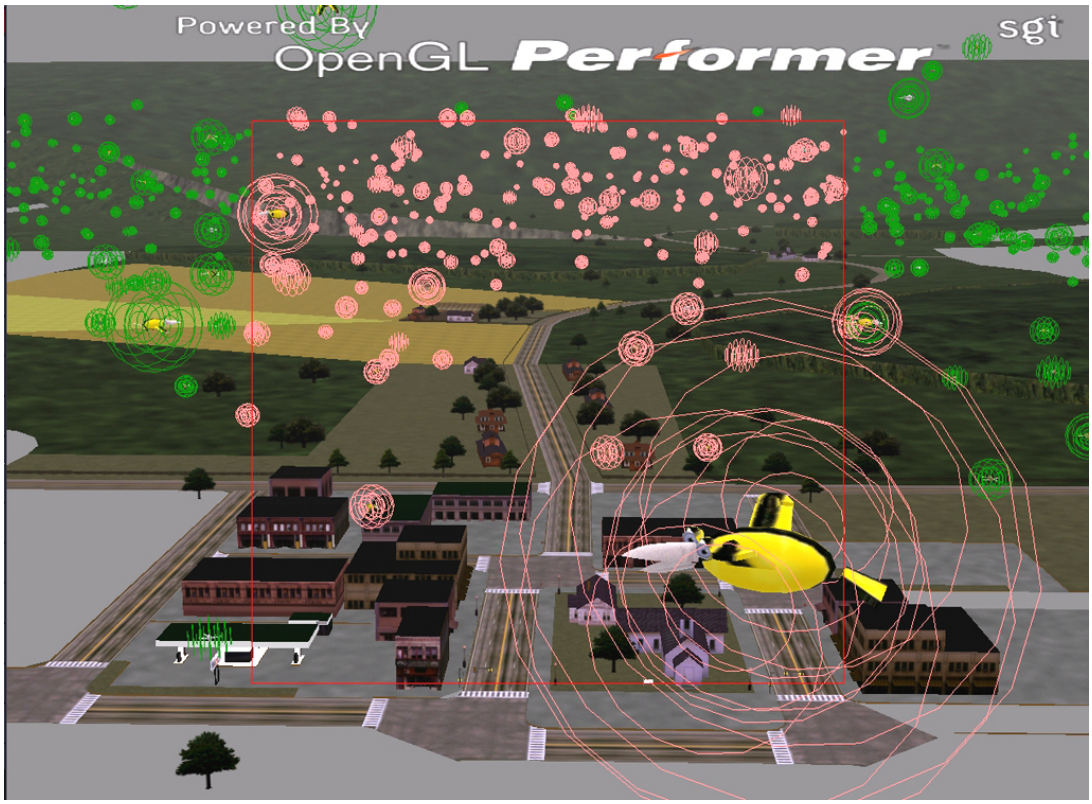


Figura 9-17. Modo de visualización de las *bounding spheres* de los actores en la simulación.

En la siguiente imagen (*Figura 9-18*), se puede observar el efecto del evento EV_REFPOINTS_FROZEN (tecla "b"), sobre la aplicación. El resultado es que todos loa actores permanecen en una posición fija adoptando una formación reticular.

Figura 9-18. Modo de visualización de los actores sin cambiar de posición y distribuidos en forma de retícula.

9.7.3 Observaciones sobre la modularidad.

Para el desarrollo de la aplicación, ha sido necesaria la creación 8 módulos, 5 de ellos encargados de definir extensiones, uno encargado de definir al actor virtual y su micro-API, otro encargado de actuar como interfaz entre la aplicación final y los actores virtuales, y uno final, encargado de actuar como interfaz entre la librería de actores y el grafo de escena. Los nombres y tamaño de estos módulos son mostrados en las siguientes tablas:

Módulos de definición de extensiones:

Módulo de gestión de posturas.	vaxPose.c (221 líneas) y vaxPose.h (19 líneas).	240 líneas.
Módulo de gestión de tablas de movimientos.	vaxKftable.c (318 líneas)y vaxKftable.h (23 líneas)	341 líneas.
Módulo de gestión del parpadeo.	vaxBlink.c (208 líneas) y vaxBlink.h (22 líneas)	230 líneas.
Módulo de control de la mirada	vaxLook.c (372 líneas) y vaxLook.h (33 líneas).	405 líneas.
Módulo de control del vuelo.	vaxFly.c (274 líneas) y vaxFly.h (33 líneas)	307 líneas.

Módulo de definición y gestión del actor virtual.	birdVa.c (867 líneas) y birdVa.h (28 líneas)	895 líneas.

Módulo de integración de los actores virtuales en la aplicación final.	birdsApp.c (333 líneas) y birdsApp.h (34 líneas)	367 líneas.

Módulo de configuración de la librería para utilización con *OpenGl Performer*.	vaInPf.c (244 líneas) y vaInPf.h (5 líneas)	249 líneas.

Adicionalmente, han sido necesaria la utilización de unos ficheros auxiliares con utilidades sobre cinemática inversa (total 468 líneas, fichero de cabecera *"ikLookFuncs.h"*), y de carga de ficheros (total 260 líneas, fichero de cabecera *"loadGeoFile.h"*).

Los 5 módulos de extensión empleados, han sido implementados con una cantidad muy reducida de líneas de código, y han sido implementados de una forma sencilla y rápida. Los módulos de *keyframing* y posturas pueden ser reutilizados en cualquier tipo de actor virtual, y los módulos de parpadeo y gestión de la mirada pueden ser reutilizados en cualquier tipo de actor que disponga de párpados, y ojos y cuello orientables. La extensión de gestión de vuelo es un poco más específica, pero, aún así, podría ser reutilizada para otro tipo de actores. Estos módulos también pueden ser ampliados internamente en el caso necesario, por ejemplo añadiendo una utilidad de cambio suave entre tablas de *keyframing*, o haciendo más sofisticada la forma de gestión del vuelo.

La utilización de módulos de extensión se ha presentado como una forma muy adecuada para implementar los modelos motor y comportamental de los actores virtuales. En este ejemplo se ha mostrado la capacidad para organizar los módulos de extensión de una forma jerárquica: Los módulos de posturas y *keyframing* han actuado como una capa de bajo nivel, y proporcionado soporte para la implementación de módulos de más alto nivel tales como las extensiones de mirada, vuelo y parpadeo.

El módulo más complejo ha sido el encargado de la definición y gestión del actor virtual (birdVa.c y .h). La definición de la topología y la geometría del actor ha sido realizada de forma manual, para lo que ha sido necesario haber realizado una planificación previa, éste ha sido uno de los aspectos que ha resultado más laborioso (en este sentido sería adecuado emplear alguna utilidad gráfica que auxiliase en el ensamblado de los actores virtuales). El hecho de que este módulo proporcione un "micro API" de funciones de acceso las funcionalidades del actor virtual, introduce una capa de abstracción muy útil, que permite que un desarrollador de aplicaciones pueda emplear este tipo de actor sin necesidad de tener que conocer nada en relación con la forma en la que el actor está definido, o a las extensiones que emplea internamente.

El módulo que configura a la librería para su utilización en *Opengl Performer* (ficheros vaInPf.c y .h), puede ser empleado en cualquier otra aplicación con actores virtuales basada en la utilización de esta librería.

El módulo de integración de los actores virtuales en la aplicación final es totalmente reconfigurable, en este caso hace que los actores se comporten de una forma orientada a la demostración de las capacidades de este trabajo, si se desease que los actores se comportasen de otro modo, bastaría con adaptar este módulo de la forma necesaria.

9.7.4 Observaciones sobre el rendimiento.

En este apartado se va emplear este ejemplo de aplicación con un número elevado de actores, para analizar de una forma práctica, cual es el rendimiento conseguido mediante la utilización de las estructuras y métodos descritos en este trabajo.

El ordenador en el que se han realizado las mediciones que serán mostradas en este apartado ha sido un PC basado en una CPU Intel P4 a 3 GHz, y una tarjeta gráfica basada en un procesador Nvidia FX5600. Como sistema operativo se ha empleado Linux RetHat versión 7.2, y la librería *OpenGl Performer* en su versión 2.5.2. Para analizar los tiempos consumidos en dibujado de cada frame se emplearan las utilidades proporcionadas por la utilidad "*perfly*". La resolución de la ventana gráfica empleada para las simulaciones ha sido de 1024x768 pixeles.

Para comprobar el rendimiento se va a definir una posición fija de la cámara, y se va a ejecutar la simulación con distintas modificaciones que permitan conocer los costes parciales de cada operación.

Para simular las condiciones medias de la aplicación, la cámara tendrá una posición central dentro de la ciudad y el grupo de actores.

A continuación se muestra una imagen que se correspondería con lo que se podría una situación normal de simulación. Dicha imagen (*Figura 9-19*), se corresponde con una simulación en la que hay un total de 2000 actores aleatoriamente sobre el cielo de la ciudad.

Figura 9-19. Escena de simulación con 2000 actores virtuales.

Consultando la información sobre rendimiento proporcionada por la utilidad "perfly", se obtiene que el tiempo medio obtener esta vista es de 18.7 milisegundos. Considerando una tasa de refresco de 50 imágenes/segundo, se dispone de un tiempo máximo de 20 milisegundos completar el dibujado de cada imagen, es decir, la simulación mostrada funciona correctamente en tiempo real consumiendo un 93 % de la potencia total de cálculo.

El resultado de ejecutar la función *vaStatsShow()* sobre esta escena es el siguiente:

```
Refpoint->   processed:  2000  matUpdated: 1820   coherency:  9.0%
Sklsroot->   processed:   250  matUpdated:  229   coherency:  8.2%
Dofs->       processed:   248
Skleletons   processed:   364  matUpdated:  360   coherency:  1.4%
```

De la observación de esa información se obtiene que de los 2000 actores que hay en escena solamente se están recalculando 1820 matrices de *Refpoint.* Esto es debido a que cuando un pájaro alcanza una posición destino, se queda esperando unos segundos hasta que le es asignado un nuevo punto al que ir.

Durante este periodo sus *RefpointMat* permanecen inalteradas, y el mecanismo de coherencia evita que se consuma tiempo en su recálculo. El 9.80% de los actores de la escena se encuentran en esta situación.

La primera fase de CULL determina que solamente 250 actores tienen sus *Refpoint Bspheres* total o parcialmente dentro del frustum. Sobre estos actores solamente ha sido necesario actualizar la matriz S*klsrootMat* de 229, es decir, un 8.2% de los actores permanecen sin modificar la posición su *Skeletons Root,* esto está ocurriendo con los actores que están en una trayectoria descendente.

La segunda fase de CULL determina que 248 actores tienen su *Sklsroot Bsphere* total o parcialmente dentro del frustum. El hecho de que la segunda fase de *CULL* solamente elimine a dos actores, se debe a que la diferencia entre los radios de las *Sklsroot Bspheres* y las *Refpoint Bspheres* de este tipo de actor es muy pequeña.

Respecto al procesado de nodos *Skeleton*, solamente se están procesando 364, un número muy inferior al que se obtendría multiplicando el número de actores dentro del frustum por su número de nodos *Skeleton*. Esto es debido a la actuación de los niveles de detalle topológicos, y al hecho de que la mayoría de los actores se encuentran muy alejados de la cámara.

De los 364 nodos Skeleton procesados, se está actualizando la matriz de 360, este valor de coherencia tan pequeño (1.4%), es debido a que este tipo de actores presenta una gran movilidad en los valores de sus grados de libertad.

El tiempo medio necesario para dibujar la misma escena sin ningún actor es de 9.1 milisegundos, de lo que se deduce que el tiempo consumido por la gestión de los actores virtuales es de 9.6 milisegundos.

El tiempo empleado en la gestión de los actores virtuales puede ser dividido en dos bloques: el tiempo consumido por el dibujado de su geometría, y el coste dedicado a su procesamiento. Para conocer cual es el tiempo consumido en el dibujado de los actores se emplea la función *vaStatsSetTestMode()* con el modo VASTATS_DISABLE_DRAWING. Esta función actúa sobre todos los nodos *vaLod* de los actores de la escena bloqueando el dibujado de las geometrías que dependen de ellos. El resultado de realizar dicha operación es una reducción del tiempo de dibujado de 3,4 milisegundos. Restando esta cantidad del coste total de gestión de los actores, se obtiene el coste de procesado de los actores virtuales de 6.2 milisegundos.

El tiempo total de procesado de los actores virtuales (6.2 milisegundos) está compuesto de distinto tipos de operaciones:

- El coste de acceso a los 2000 nodos Actor que componen la escena, comprobación de CULL con sus *Refpoint Bspheres,* y procesado de *SklsRoots* y CULL con *Sklsroot Bspheres* (1.9 msecs). Se puede considerar también como formado por tres costes parciales:
    1. Traverse de los 2000 nodos *Actor* existentes en la escena (0.9 msecs).

2. Comprobación de CULL con sus *Refpoint Bspheres* (0.6 msecs).
3. Procesado de *Sklsroots* y comprobación de CULL con *Sklsroot Bspheres* (0.4 msecs).

- El coste de gestión del comportamiento (1.1 msecs). Está formado por 3 costes parciales:
  1. Gestión de comportamiento asociado al *Reference Point* (0.2 msecs).
  2. Gestión de comportamiento asociado al *Skeletons Root* (0.5 msecs).
  3. Gestión de comportamiento asociado a los grados de libertad (0.4 msecs).
- Recorrido de los nodos que componen internamente al actor, y generación de matrices (2.2 msecs). Formado por dos costes parciales.
  1. Generación de las matrices de los nodos *Skeleton* (0.8 msecs)
  2. Traverse de los nodos *Skeleton*, *LOD*, etc. empleados por los actores (1.1 msecs).
- Aplicación de matrices de los nodos *Actor* y *Skeleton* (1 msec).

Estos costes parciales han sido obtenidos mediante la utilización de la función *vaStatsSetDisableMode()* con sus distintos parámetros. La importancia relativa de cada tipo de operación puede ser observada claramente en la *Figura 9-20* mostrada a continuación.

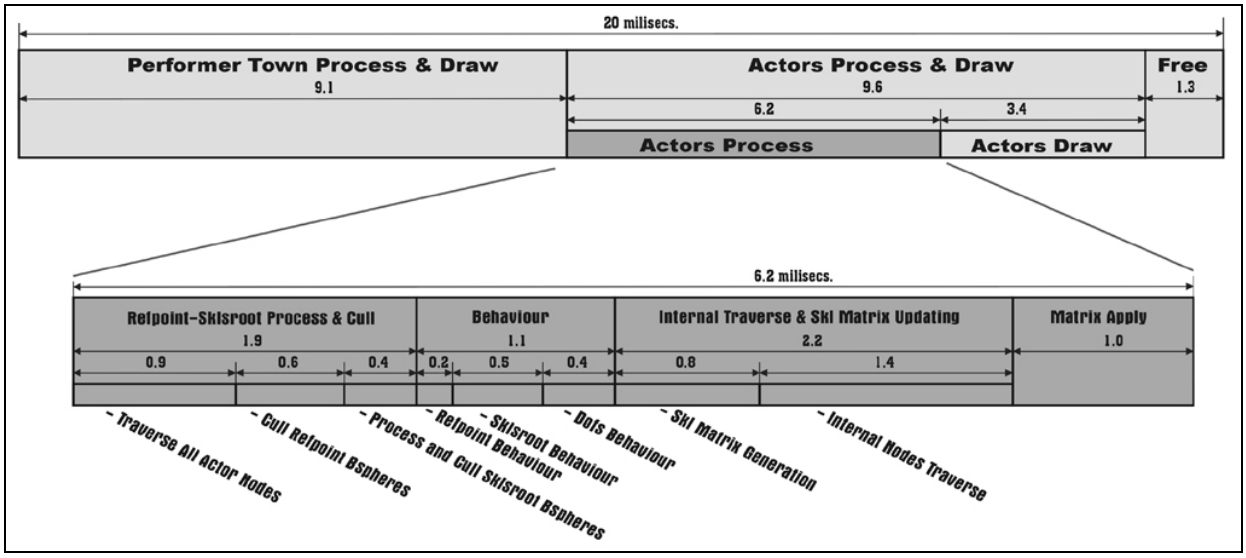


Figura 9-20. Costes parciales del procesado de la escena ejemplo con 2000 actores virtuales.

Las estructuras y métodos propuestos en este trabajo actúan reduciendo al máximo los costes asociados al procesado de Actores (*Actor Process* en la *Figura 9-20*). El coste etiquetado como *Actors Draw*, está relacionado con el número de polígonos necesarios para dibujar a los actores virtuales, y su coste sería similar si se hubiese empleado un grafo de escena tradicional.

Se puede observar que la gestión de comportamiento (etiquetada en la figura como *Behaviour*) tiene un coste relativamente bajo, esto es debido a que los mecanismos de gestión de comportamiento de este ejemplo de aplicación son muy sencillos. También llama la atención el hecho de que la gestión de comportamiento vinculada con la actualización de los *DOFs* (etiquetada como *Dofs Behaviour*), tenga un

coste equivalente al del coste vinculado a la gestión de comportamiento del *Skeletons Root* (etiquetada como *Refpoint Behaviour*), eso es debido a la gestión de nivel de detalle de comportamiento está actuando de forma adecuada.
El elevado coste del procesado y CULL de *Refpoints* y los *Sklsroots* (etiquetado como *Refpoint-Sklsroot Process &Cull*), es debido principalmente a que existe unas cuantas operaciones básicas que han de ser realizadas sobre la totalidad de los actores de la escena, con el consiguiente consumo temporal.

Una tasa de refresco de 50 Hz es muy exigente, en muchos casos una tasa de actualización de 25 imágenes/segundo presenta una calidad suficiente. Se ha probado cuantos actores sería posible simular con este nuevo requisito, obteniéndose que es posible simular de forma adecuada 6000 actores (representado en la *Figura 9-21*).

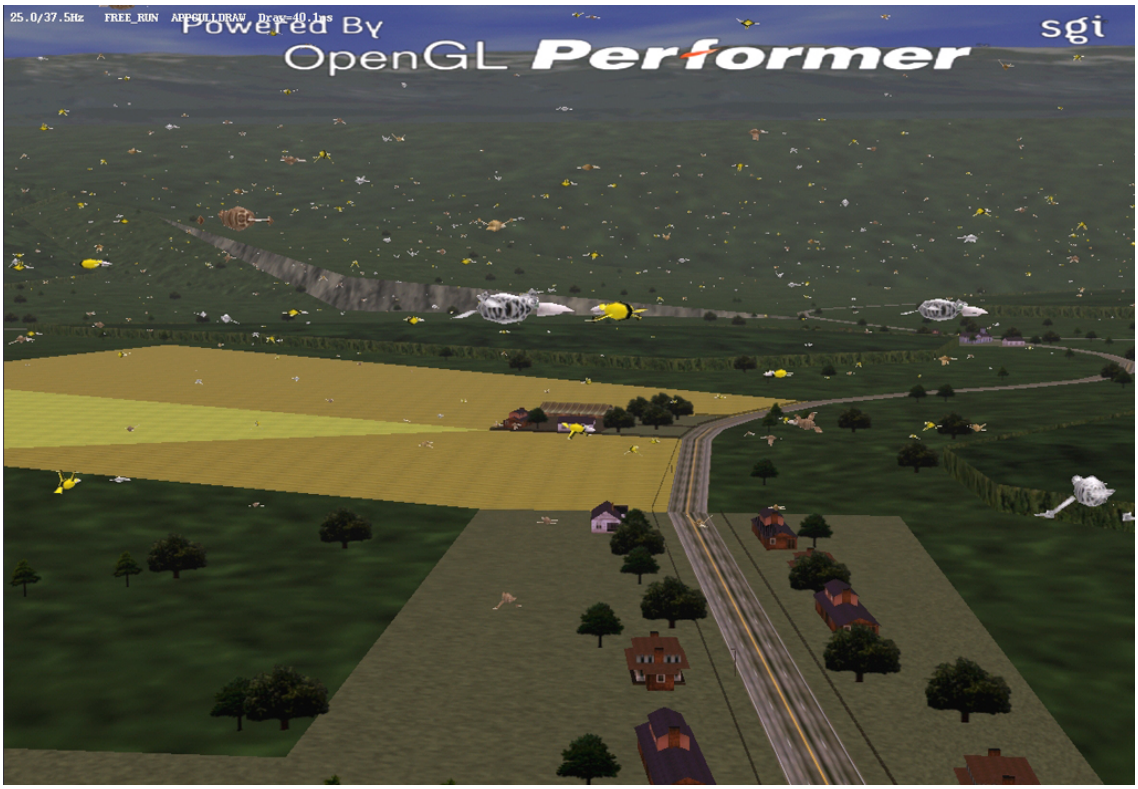


Figura 9-21. Escena de simulación con 6000 actores y una tasa de actualización de 25 imágenes/segundo.

9.8 Conclusiones.

En este capítulo se ha mostrado como las estructuras y métodos propuestos en esta tesis son totalmente aplicables en una situación real. Para ello se ha empleado la librería y arquitectura descritas en el capítulo anterior, para integrar una gran cantidad de actores en una aplicación de simulación ya existente. Para realizar esta integración se han empleado la mayoría de las funciones de la librería, constituyendo un ejemplo suficientemente significativo para demostrar su fiabilidad.

Este capítulo puede parecer muy largo, pero no lo es tanto si se tiene en cuenta que ha sido posible explicar totalmente (incluyendo el código), todos los aspectos de la integración de actores virtuales en una aplicación de simulación, yendo desde la definición de las geometrías hasta la incorporación de módulos de comportamiento, la independencia del grafo de escena, y la integración en aplicación ya existente. El contenido de este capítulo constituye un buen ejemplo que puede ayudar a futuros desarrolladores en la implementación de este tipo de aplicaciones.

La utilización de la arquitectura modular propuesta este ejemplo de aplicación ha demostrado sus ventajas organizativas. Para la implementación de subsistema de gestión de actores ha sido necesario un total de 3000 líneas de código divididas en 8 módulos (cinco definiendo extensiones, uno definiendo al actor virtual, un módulo que actúa de interfaz con el grafo de escena y otro que actúa como interfaz con la aplicación principal). Los módulos de extensión han sido implementados de una forma muy rápida, su integración ha resultado muy sencilla, y sobre todo, el resultado obtenido es totalmente reutilizable para futuros desarrollos. La definición de la topología y las geometrías del actor de una forma manual, ha sido la parte más laboriosa, evidenciando la necesidad de alguna utilidad interactiva que facilite el ensamblado del actor virtual

La arquitectura modular propuesta, y el hecho de módulos de extensión puedan ser organizados de forma jerárquica (existen módulos de alto nivel como el de vuelo y parpadeo, que utilizan a otros de más bajo nivel como los de *keyframing* y poses) proporciona una forma muy adecuada de implementación de los modelos motor y comportamental.

Respecto al rendimiento del sistema, se ha demostrado que es posible integrar 2000 actores en la aplicación "perfly + Performer Town", con una tasa de refresco de 50 imágenes/segundo, o 6000 actores con una tasa de refresco de 25 imágenes/segundo, empleando para ello un ordenador doméstico.

La elección del grafo de escena proporcionado por *OpenGl Performer,* la integración sobre su aplicación más popular y su database más conocida, el alto rendimiento obtenido, y la capacidad de organización modular del trabajo demuestra claramente la capacidad de aplicación práctica de los resultados de este trabado doctoral.

PARTE V

CONCLUSIONES

Capítulo 10. Conclusiones, contribuciones y trabajo futuro.

10.1 Índice.

10.1 Conclusiones.

Esta Tesis Doctoral esta centrada en la búsqueda método estándar para la definición e integración de un conjunto elevado de actores virtuales en una aplicación de simulación. Para ello se ha comenzado realizando un estudio sobre el estado del arte de la informática gráfica en tiempo real, y también de las investigaciones y desarrollos relacionados con los actores virtuales. La informática gráfica en tiempo real actual está caracterizada por la utilización de hardware especializado en la generación de imágenes, el cual es gestionado mediante librerías gráficas de bajo nivel (como *OpenGl*), y también por la utilización de librerías de alto nivel basadas en un *grafo de escena* (como *OpenGl Performer*), para el desarrollo de escenas de simulación complejas. La gran mayoría de las aplicaciones de simulación actuales están desarrolladas en torno a un grafo de escena, y éste constituye, por tanto, el substrato básico sobre el cual realizar las aportaciones que sean necesarias para el soporte de los actores virtuales. En este sentido se han analizado las características comunes a los grafos de escena actuales, y se ha tomado la decisión de que las estructuras y métodos aportados sean susceptibles de ser aplicados en cualquier grafo de escena que presente unos requisitos mínimos.

Respecto al estado del arte de los actores virtuales, éste ha estado caracterizado por la utilización de arquitecturas multicapa (comportamental, motora, geométrica), y la existencia de varios campos de aplicación que evidencian el enorme potencial de aplicación de este tipo de tecnología. La conexión entre las estructuras y métodos empleados para implementar actores virtuales, y la forma de operar de los grafos de escena ha sido mínima. La gran mayoría de las investigaciones han estado centradas actores de tipo humanoide, y especialmente el modelado comportamental, prestando poco atención a los modelos motor y geométrico. También ha existido una tendencia a crear los actores virtuales y sus mundos, siguiendo una filosofía "Top-Down", según la cual, se ha modelado primero su inteligencia, luego un cuerpo que pudiese mostrar las actividades ideadas por esa inteligencia, y por último un entorno 3D en el que ese cuerpo pudiese realizar sus actividades. Esta orientación es la causa de que la gran mayoría de actores virtuales actuales vivan encerrados en sus propias aplicaciones, y, además, no coincide con la realidad de los desarrolladores de aplicaciones de simulación, los cuales disponen de un substrato estable para desarrollar sus escenarios 3D (mediante grafos de escena), y desean, en primer lugar, disponer de actores virtuales que presenten una representación geométrica compatible con esos entornos, y, en segundo lugar, dotar a esos actores de la mayor capacidad de inteligencia posible. En este trabajo se ha seguido esta segunda orientación, proporcionando nuevas estructuras y métodos que, agregados sobre una estructura de *grafo de escena*, permitan definir el modelo geométrico de los actores virtuales de un modo estandarizado y computacionalmente eficiente. Las estructuras y métodos propuestos han sido diseñados de modo que proporcionen, además, un buen substrato para el desarrollo de los modelos motor y comportamental.

El principal problema de los grafos de escena actuales en relación con los actores virtuales es que han sido ideados para representar entornos estáticos, o con objetos que presenten movimientos simples, y

presentan serias carencias a la hora de representar objetos con una estructura articulada compleja, o elementos cuya gestión de comportamiento tenga un coste computacional elevado. Se ha demostrado la poca adecuación de los grafos de escena actuales para la definición de la estructura articulada de un actor virtual, y se han presentado dos nuevos tipos de nodos especialmente orientados a la definición y gestión de actores sintéticos en tiempo real. El diseño de estos nuevos nodos (*Actor* y *Skeleton*), ha sido realizado prestando especial atención a la estandarización y a la eficiencia computacional. Mediante ellos es posible definir cualquier tipo de estructura articulada, y además están interrelacionados de tal modo que encapsulan la complejidad jerárquica del actor virtual al usuario, el cual sólo tiene que aplicar ordenes de alto nivel sobre el nodo *Actor*.

También se ha mostrado, como la integración de actores virtuales en una aplicación de simulación no puede ser completada mediante la simple adición de nuevos nodos sobre un grafo de escena. La complejidad de sus modelos motor y comportamental, el hecho de estos puedan ser compartidos por varios actores, su necesidad de extensibilidad, o de ser capaces de actuar en simulaciones macroscópicas, hace necesaria alguna estructura adicional que se encargue de almacenar las características de comunes y de alto nivel de una determinada "especie" de actores. La estructura *"Clase de Actor"* (o *vaActclass*) proporciona estas capacidades, actuando como complemento necesario a los nodos *Actor* y *Skeleton:* Los nodos *Skeleton* y *Actor* proporcionan un método rígido, sencillo, estable, eficiente para integrar un actor virtual en un grafo de escena; La estructura *"Clase de Actor"* proporciona una estructura flexible, reutilizable y ampliable, capaz de almacenar todos las informaciones de alto nivel que puedan ser necesarias para la definición de un actor virtual complejo.

De forma adicional a la definición de los nodos *Actor* y *Skeleton*, y a la estructura *vaActclass*, se han tratado varios aspectos del procesado de un grafo de escena, que necesitaban ser adaptados adecuadamente a las características de una simulación con múltiples actores virtuales. Estos aspectos han sido: la forma en la que se procesan las matrices de transformación, la forma en la que se realiza el culling, el modo de gestión de los niveles de detalle, y integración en un sistema multiprocesador. En relación con las matrices de transformación, se ha mostrado la facilidad con la que actúan como cuello de botella, y se ha actuado en consecuencia, haciendo que los nuevos nodos propuestos, y los nuevos métodos de gestión de culling y LOD, actúen reduciendo al mínimo el coste de este tipo de operaciones, y aprovechando la capacidad del hardware gráfico para realizar estas operaciones. En relación con la gestión de *culling*, se ha propuesto un nuevo método específico para actores virtuales, que es compatible con el tradicional, minimiza las operaciones de recálculo de bounding spheres, emplea el hardware de multiplicación de matrices, y actúa sobre los costes asociados a la gestión del comportamiento. En relación con la gestión de nivel de detalle, la gestión tradicional de tipo geométrico encaminada a reducir el número de polígonos enviados al hardware gráfico, ha sido complementada con una gestión de nivel de detalle *topológico*, que consigue que el número de articulaciones empleadas por un actor disminuya con la distancia, y un nivel de detalle *comportamental,* que permite que los métodos de gestión de comportamiento empleados también reduzcan su coste con la distancia. Por último, en relación con la integración de los actores virtuales en un sistema multiprocesador, se ha mostrado como la división

clásica del trabajo en tres etapas APP, CULL, DRAW, ejecutadas en forma de pipeline, no resulta adecuada para una simulación con actores virtuales, y se han propuesto alternativas adecuadas.

Se ha dedicado un capítulo a estimar de forma teórica la mejora computacional introducida por la utilización de las estructuras y métodos descritos en este trabajo. El resultado ha sido la obtención de varias expresiones que permiten observar la importancia relativa de las diversas operaciones elementales realizadas, estimar el coste de procesado en función de distintos parámetros, y también la mejora computacional en relación con un procesado en un grafo de escena tradicional. En este análisis se muestra como las nuevas estructuras y métodos hacen que el procesado de actores se realice entre 50 y 120 veces más rápido que en el caso de haber empleado un grafo de escena tradicional.

Con el objeto de demostrar la utilidad práctica de los resultados de este trabajo, se ha implementado una librería en C que proporciona acceso a las estructuras y métodos descritos, y, además, se ha propuesto una arquitectura modular que contempla todos los aspectos de la integración de actores virtuales en una aplicación de simulación. Para concluir, se ha desarrollado un ejemplo de aplicación, en el cual se han empleado dicha librería y arquitectura, para integrar una gran cantidad de actores en una aplicación de simulación ya existente. En dicho ejemplo, se ha ampliado la utilidad de visualización de bases de datos proporcionada por *OpenGl Performer* (conocida como *"perfly"*), de tal modo que permita la visualización simultanea de una pequeña ciudad *("Performer Town"*), y 2000 actores con una tasa de refresco de 50 imágenes/segundo, demostrándose de este modo, además de su eficiencia computacional, su capacidad de integración sobre un grafo de escena ya existente, su adecuación para implementar los modelos motor y comportamental, y su capacidad de extensión.

10.2 Contribuciones.

Las principales contribuciones de esta Tesis Doctoral pueden ser concretadas en los siguientes puntos:

- Un análisis sobre el estado del arte de las investigaciones y aplicaciones de informática gráfica tiempo real basadas en actores virtuales.

- Una propuesta estandarizada para la construcción actores virtuales de todo tipo (no solamente humanoides), que puede ser empleada en varios *Grafos de escena*, y que se basa en la ampliación de los grafos de escena actuales con dos nuevos tipos de nodos (nodo *Actor* y nodo *Skeleton*). Para la definición de estos nodos se ha tenido en cuenta la estandarización, la simplicidad, la eficiencia computacional, y el control de alto nivel.

- Una estructura paralela al grafo de escena que permite separar la información lógica, de la información de dibujado (estructura "Clase de Actor"), y que además actúa como soporte para futuras ampliaciones, y para la realización de simulaciones macroscópicas.

- Un método de extensión de las propiedades de los actores virtuales, que permite que distintos investigadores integren sus desarrollos sobre un substrato estable.

- Una descripción detallada sobre la problemática de la multiplicación de matrices que evidencia la necesidad de que el hardware gráfico tenga capacidad para realizar este tipo de operación, y hace que la forma de funcionamiento de los grafos de escena actuales haya de ser modificada. Ambos aspectos han sido tenidos en cuenta por las estructuras y métodos aportadas por este trabajo.

- Un nuevo método de gestión de culling específico para actores virtuales que minimiza el de operaciones de recálculo de bounding spheres, el número de operaciones con matrices, actúa sobre los costes asociados a la gestión del comportamiento, y además, resulta compatible con los métodos de culling tradicionales.

- Un nuevo método de gestión de nivel de detalle orientado a actores virtuales, que no sólo actúa sobre la geometría, sino también sobre la topología (nivel de detalle geométrico), y sobre el coste de la gestión del comportamiento (nivel de detalle comportamental), y adicionalmente reduce el número de operaciones de cálculo de distancias y de operaciones con matrices.

- Un análisis de la inadecuación de los modelos en los que la gestión del comportamiento es realizada en una fase previa a procesado del grafo de escena, presentando una modificación de la organización tradicional APP-CULL-DRAW que está adaptada a los requerimientos de los actores virtuales, y que es susceptible de ser empleada en un sistema multiprocesador.

- Un análisis sobre cuales son las operaciones elementales implicadas en el procesado de una escena con actores virtuales. Dicho análisis ha permitido observar la importancia relativa de cada operación, y estimar de una forma teórica la mejora computacional introducida por las nuevas estructuras y métodos.

- Una implementación de un prototipo de librería en C, formada por 134 funciones que proporcionan acceso a un total de 10 clases. Dicha librería, permite comprobar la fiabilidad y eficiencia de las estructuras y métodos aportados por esta Tesis Doctoral.

- Una propuesta de una arquitectura modular que aborda el problema de la integración de los actores en una aplicación de simulación desde un punto de vista global.

- Un ejemplo de aplicación, basado en utilización de la librería y arquitectura propuestas, que muestra en detalle todos los aspectos de la integración de actores virtuales en una aplicación de simulación ya existente.

10.3 Trabajo futuro.

Este trabajo propone una capa de nivel medio que proporciona un método genérico de descripción y gestión de actores virtuales en tiempo real. Una vez definido este substrato básico, se observan distintos aspectos que pueden ser objeto de futuras líneas de investigación:

- Descripción de un formato de fichero estándar para la descripción de actores virtuales, que recurra al empleo de las estructuras propuestas en esta tesis, y que presente la capacidad de integrar formatos de información dispares (dlls de gestión de comportamiento, tablas de keyframing, tablas de posturas, sonidos...).

- Descripción de un método formal para la definición de los módulos de extensión. Búsqueda de un sistema de organización jerárquica de controles de comportamiento, que permita integrar y organizar los distintos métodos de control que pueden estar actuando simultáneamente sobre un mismo actor virtual.

- Realización de módulos de configuración de la librería de actores, para diversos de grafos de escena diferentes de OpenGl Performer (por ejemplo VTK, OpenSG, osg, etc.).

- Profundizar en el estudio de la realización de simulaciones macroscópicas, buscando distintos métodos de generación automática de actores virtuales.

- Búsqueda de un método que permita añadir de capas de información semántica sobre el grafo de escena, y que facilite a los actores virtuales autónomos la percepción del entorno en el que realizan sus actividades.

- Búsqueda de algún método que permita la definición de cuerpos con piel continua (skining), de una forma estándar, compatible con los requerimientos de la simulación en tiempo real, y con la forma de actuación de los grafos de escena actuales. Estos modelos podrían ser empleados en los niveles de detalle más cercanos.

- Creación de alguna herramienta de modelado que facilite la definición de la estructura articulada del actor virtual y sus niveles de detalles topológicos y geométricos.

- Creación de alguna utilidad que actúe como banco de pruebas del comportamiento de los actores virtuales, en una fase previa a su integración en la aplicación final de simulación.

- Creación de una versión evolucionada de la estructura "Clase de Actor" en la que existan mecanismos de herencia entre especies (así por ejemplo la clase de actor encargada de gestionar un actor "humano" podría beneficiarse de ciertos métodos implementados para la clase "mamífero").

Por otro lado, el hecho de que los actores virtuales sean entes tan complejos, hace que sobre el substrato básico propuesto en este trabajo, puedan ser añadidas, con objetivos muy diversos, una gran multitud de capas de más alto nivel. En este sentido existen futuras líneas de trabajo consistentes en la implementación de diferentes módulos de extensión que puedan ser integrados en la arquitectura modular propuesta. Algunos ejemplos de este tipo de integración podrían ser:

- Métodos de control de la locomoción.
- Métodos de control de la atención.
- Mecanismos de percepción de entorno mediante visión artificial y otras técnicas.
- Distintos tipos de modelos de control cinemática y dinámico.
- Mecanismos de descripción de comportamiento basados en reglas, máquinas de estados, inteligencia artificial, etc.

PARTE VI

REFERENCIAS

Capítulo 11. Referencias.

-[AIR90] John M. Airey, John H. Rohlf, and Frederick P. Brooks. *"Towards image realism with interactive update rates in complex building enviromments"*. Computer Graphics (Proceedings 1990 Symposium on Interactive 3D Graphics), 24(2):41-50, Marzo 1990.

-[ALL02] Allen B., Curless B., y Popovic Z. *"Articulated body deformation from range scan data"*. Proceedings of ACM SIGGRAPH'02. pp.612-819. 2002.

-[ALL03] Brett Allen, Brian Curless y Zoran Popovic. *"The space of human body shapes: reconstruction and parameterization from range scans"*. Proceedings of ACM SIGGRAPH´03. pp.587-594. 2003.

-[ARI02] Arikan O., y Forsythe D. *"Interactive motion generation from examples"*. Proceedings of ACM SIGGRAPH'02. pp.483-490. 2002.

-[ASH01] K. Ashida, S.-J. Lee, J. Allbeck, H. Sun, N. Badler, y D. Metaxas. *"Pedestrians: Creating agent behaviors through statistical analysis of observation data"*. Proc. Computer Animation 2001.

-[AZU95] Ronald Azuma, Gary Bishop, "*A Frequency-Domain Analysis of Head-Motion Prediction"*, Proc. ACM SIGGRAPH'95. 1995.

-[BAD93] N. Badler, C. Phillips, and B. Webber. "*Simulating Humans"*: Computer Graphics Animation and Control. Oxford University Press, New York. 1993.

-[BAT94] J. Bates, *"The role of Emotion in Believable Agents"*, Communications of the ACM, vol 37, no. 7, pp.122-125. 1994.

-[BAD96] N.Badler, B. webber, W. Becket, C. Geib, M. Moore, C. Pelachaud, B. reich,y M. Stone. *"Planning for animation"*. Computer Animation. Prentice-Hall. 1996.

-[BAD97] N.I. Badler, B.D. Reich y B.L. Webber, *"Towards Personalities for Animated Agents with Reactive and Planning Behaviours"*, Creating Personalities for Synthetic Actors, Trappl R. and Petta P. Eds., Springer-Verlag, pp. 43-57. 1997.

-[BAD98] N. Badler. "*Virtual Humans for Animation, Ergonomics, and Simulation*." Centr for Human Modeling and Simulation. University of Pennsylvania.1998.

-[BAD99] Norman I. Badler, Martha S. Palmer y Rama Bindiganavale. *"Animation Control for Real-Time Virtual Humans"*. Communications of the ACM, 42(8). pp.64-73. Agosto 1999.

-**[BADa02]** N.Badler and J.M. Allbeck.*"Embodied autonomous agents. Handbook fof Virtual Environments"*. K.Stanney, Ed., Lawrence Erlbaum Associates. pp. 313-332. 2002.

-[BADb02] N. Badler, C. Erignac, y Y. Liu. *"Virtual humans for validating maintenance procedures"*. Comm. of the ACM, Vol. 45, Issue 7. pp.56-63. Julio 2002.

-[BAR98] David Baraff y Andrew Witkin. *"Large steps in cloth simulation".* Siggraph98 Conference Proceedings. pp. 43-54. 1998.

-[BAT94] J. Bates. "*The role of emotion in believable agents"*. Comm. of the ACM, 37(7) pp.122-125. 1994.

-[BAY95] Bayarri, S. *"Técnicas de Visualización y Simulación en Tiempo Real de Entornos de Conducción, Nuevos Algoritmos, Estructuras de Datos y su Gestión"*. Tesis Doctoral. Universidad de Valencia. 1995

-**[BDI96]** Boston Dinamics Inc. "*BDI-Guy software system"*. 1996.

-[BEA95] C. Beardon y V. Ye. *"Using Behavioural Rules in Animation"*, Computer Graphics: Developments in Virtual Environments, R. A. Earnshaw, J. A. Vince (eds). Academic Press Ltd, London, pp.217-234. 1995.

-[BEC97] W. Becket. "*Reinforcemente Learning of reactive Navigation for Computer Animation of Simulated Agents".* Tesis Doctoral, CIS, University of Pennsylvania. 1997.

-[BEE90] R.D. Beer. *"Intelligence as Adaptative Behaviour"*, Academic Press Ltd. 1990.

-[BER97] F. Scheepers, R. Parent, W. Carlson, y S. May. *"Anatomy-Based Modeling of the Human Musculature"*. Proceedings of ACM SIGGRAPH'00. pp.163-172. 1997.

-[BLA99] Volker Blanz, Thomas Vetter. *"A Morphable Model For The Synthesis Of 3D Faces"*. Proceedings of ACM SIGGRAPH'99. pp.187-194. 1999.

-[BOU90] Boulic R., Magnenat-Thalmann N. M.,Thalmann D. "*A Global Human Walking Model with Real Time Kinematic Personification"*., The Visual Computer, Vol.6(6). 1990.

-[BOU95] Boulic R., Capin T., Huang Z., Kalra P., Lintermann B., Magnenat-Thalmann N., Moccozet L., Molet T., Pandzic I., Saar K., Schmitt A., Shen j., Thalmann D. "*A system for the Parallel Integrated Motion of Multiple Deformable Human Characters with Collision Detection"* , Computer Graphics Forum EUROGRAPHICS'95 Proceedings , pp. 337-348, Maastricht.1995.

-[BLU97] B.M. Blumberg y T.A. Galyean, *"Multi-level Control for Animated Autonomous Agents: Do the Right Thing...Oh, Not That"*, Creating Personalities for Synthetic Actors; Trappl R. and Petta P. Eds., Springer-Verlag, pp. 74-82. 1997.

-[BLU02] Blumberg, Downie, Ivanov, Berlin, Johnson, Tomlinson. *"Integrated learning for interactive synthetic characters"*. Proceedings of ACM SIGGRAPH'02. pp.417-426. 2002.

-[BOU97] Roman Boulic, Pascal Bécheiraz, Luc Emering, Daniel Thalman. "*Integration of Motion control Techniques for Virtual Human and Avatar Real-Time Animation"*. Proc. ACM International Symposium VRST'97, pp.111-118. 1997.

-[BRA01] Brand M. *"Morphable 3D models from video"*. Proceedings of the IEEE conference on Computer Vision and Pattern Recognition(CVPR). pp.456-463. 2001.

-[BRO92] Brown, R. G. and P. Y. C. Hwang. "*Introduction to Random Signals and Applied Kalman Filtering"*, 2nd Edition, John Wiley & Sons, Inc. 1992.

-[BRU95] A. Bruderlin y L. Williams. "*Motion signal processing In Computer Graphics"*, Annual Conf. Seriers, pp. 97-114. ACM 1995.

-[BRU96] Armin Bruderlin and Lance Williams. "*Motion Signal Procesing"*, Siggraph96 conference Proceedings. pp. 97-104. 1995.

-[CAP97] T. Capin, M. Jovovic, J. Esmerado, A. Aubel, D. Thalmann. "*Efficient Network Transmission of Virtual Human Bodies"*. Computer Graphics Laboratory, Swiss Federal Institute of Technology.1997.

-[CAP97] T.K. Capin, H. Noser, D. Thalmann, I.S. Pandzic, N. Magnenat Thalmann. "*Virtual Human Representation and Commnuncation in VLNET Networked Virtual Enviromment"*, IEEE Computer Graphics and Applications. 1997.

-[CAP98] T. K. Capin, I. Sunday Pandzic, N. Magnenat Thalmann, D. Thalmann. "*Realistic Avatars and Autonomous Virtual Humans in VLNET Networked Virtual Enviromments"*. Computer Graphics Laboratory Swiss Federal Institute of Technology. Laussane, Switzerland.1998.

-[CAP00] T. K. Capin, E. Petajan y J. Ostermann. *"Efficient modeling of virtual humans in MPEG-4"*. IEEE International Conference On Multimedia And Expo (ICME), New York, NY, Volume: 2, pp. 66-75. Febrero 2003.

-[CAP02] Capell S., Green S., Curless B., Duchamp, T., y Popovic Z. *"Interactive skeleton-driven dynamic deformations"*. Proceedings of ACM SIGGRAPH´02. pp.586-593. 2002.

-[CAS94] J. Cassel, C. Pelachaud, N.Badler, M.Steedman, B.Achorn, W.Becket, B.Douville,S.Prevost, y M. Stone. "*Animated conversation:Ruled-Based generation of facial expresion, gesture and spoken intonation for multiple conversational agents*." In Computer Graphis, Annual Conf. Series, pp. 413-420. ACM, 1994.

-[CAT78] Catmull, E.; Clark, J. "*Recursively Generated B-spline surfaces on arbitrary topological meshes"*. Computer Aided Design vol. 10 no. 6, pp. 350-5. 1978

-[CHA89] Chadwick J.E., Haumann D.R., Parent R.E., "*Layered Construction forDeformable Animated Characters"*, Proceedings Siggraph'89, 23, 3, pp. 243-252. 1989.

-[CHA02] Chang J., Jin J., y Yu Y. *"A practical model for hair mutual interactions"*. Proceedings of the ACM SIGGRAPH Symposium on Computer Animation. pp.73–80. 2002.

-[CHO95] S. Chopra. "*Strategies for simulating direction of gaze and attention"*. technical report, Center for Human Modelling and Simulatión, University of Pennsylvania. 1995.

-[CHO99] S. Chopra-Khullar y N. Badler. *"Where to look? Automating visual attending behaviors of virtual human characters"*. Autonomous Agents Conf., Seattle, WA. Mayo 1999.

-[COQ90] Coquillart S., Extended Free Form Deformation : "*A Sculpturing tool for 3d geometric Modeling"*, Proceedings Siggraph'90, 24, 2 , pp. 187-193. 1990.

-[CREM96] J. Cremer, J. Kearney y H. Ko, *"Simulation and Scenario support for virtual environments"*, Computer & Graphics, vol. 20, no. 2 , pp.199-206. 1996.

-[CULL98] Culler ,David E. , y Jaswinder Pal Singh, Anoop Gupta, "Parallel Computer Architecture: A Hardware/Software Approach". Morgan Kaufmann Publishers Inc., San Francisco 1998.

-[DAS99] Dasgupta A., y Nakamura Y. *"Making feasible walking motion of humanoid robots from human motion capture data"*. Proc.IEEE Intl. Conference on Robotics and Automation. 1999.

-[DEL93] Delinguette H., Watanabe H., Suenaga V., "*Simplex Based Animation",* Proceedings Computer Animation'93. pp. 13-28. 1993.

-[DER98] Tony DeRose, Michael Kass, Tien Truong. "*Subdivision Surfaces in Character Animation"*. Siggraph98 Conference Proceedings. pp. 85-94. 1998.

-[DIR02] *"DIRECTX reference"*. DIRECTX, 2002. Microsoft DirectX SDK Documentation http://www.microsoft.com/directx. 2002.

-[DMS96] Defense Modeling and Simulation Office, *"High Level Architecture Rules Version 1.0"*, U.S. Department of Defense. 1996

-[DOE97] Peter K. Doenges, Tolga K. Capin, Fabio Lavagetto, Joern Ostermann, Igor S. Pandzic, Eric D. Petajan. "*MPEG-4: Audio/Video & Synthetic Graphics/Audio for Mixed Media"*. Published in Image Communications Journal, Vol.5, No.4. Mayo 1997.

-[DOU96]B. Douville, L. Levison, y N.N.Badler. "*Task level object grasping for simulated agents"*. Presence, 5(4) pp. 416-3430. 1996.

-[EKM78] P. Ekman and W. Friesen. "*Facial Action Coding System"*. Consulting Psychologists Press, Inc., Palo Alto, CA. 1978.

-[ESC99] M.Escher, T.Goto, S.Kshirsagar, C.Zanardi, N.Magnenat Thalmann, "*User Interactive MPEG-4 Compatible Facial Animation System"***,** International Workshop on Synthetic - Natural Hybrid Coding and Three Dimensional Imaging (IWSNHC3DI'99), Santorini, Grecia .pp.29-32. 1999.

-[FAN03] Anthony C. Fang, Nancy S. Pollard. *"Efficient Synthesis of Physically Valid Human Motion"*. Proceedings of ACM SIGGRAPH´03. pp.417-416. 2003.

-[FAL94] L. Falkenhagen. "*Depth Estimation from Stereoscopic Image Pairs Assuming Piecewise Continuous Surfaces"*. Image Processing for Broadcast and Video Productions, Springer, Great Britain, Y. Parker y S. Wilbur Eds., pp.115-127. 1994.

-[FAL01] Faloutsos P., Van de Panne M., y Terzopoulos D. *"Composable controllers for physics-based character animation"*. Proceedings of ACM SIGGRAPH'01. pp.251-260. 2001.

-[FEI96] Feijo B. CostaM. "*An architecture for concurrente reactive agentes in real-time animation".* Proc. del SIBGRAPHI IX. pp. 281-288. 1996.

-[FER98] Fernandez, M. *"Arquitectura y Desarrollo Software de Sistemas para la Modelización del Tráfico en Simulación de Conducción".* Tesis Doctoral. Universidad de Valencia. 1998.

-[FOL91] James D. Foley, Andries van Dam, Steven K.Feiner, y John F.Hughes, *"Computer Graphics Principles and Practice"*. Addisson-Wesley. 1991.

-[FOX96] E. Foxlin, "*Inertial Head-Tracker Sensor Fusion by a Complementary Separate-Bias Kalman Filter",* Proc. IEEE VRAIS'96. 1996.

-[FUC80] H. Fuchs, Z.M.Kedem, and B.F.Naylor. *"On visible surface generation by a priori tree structures"*. Computer Graphics (SIGGRAPH´80 Proceedings), 14(3):124-133, July 1980.

-**[FUN93]** Funkhouser, T.A. y otros.. *"Adaptative display algorithm for interactive frames rates visualization of complex virtual enviroments".* Proc. ACM SIGGRAPH, 1993.

-[FUN99] Funge J., Tu X., y Terzopoulos D. *"Cognitive modeling: Knowledge, reasoning and planning for intelligent characters"*. Proceedings of ACM SIGGRAPH'99. pp.29-38. 1999.

-[FUN00] J. Funge*, "Cognitive modelling for games and animation"*, Communications of the ACM, vol. 43, no. 7, pp. 40-48. 2000.

-[GAL00] Rafael Galvão João, Paulo Garcia Martins, and Mário Rui Gomes. *"Modeling reality with simulation games for a cooperative learning"*. *Proceedings of the Winter Simulation Conference (ACM*). 2000.

-[GLE98] Michael Gleicher. "*Retargeting motion to new characters"*. Siggraph98 conference Proceedings. pp. 33-42. 1998.

-[GKS88] International Standars Organization. *"International standard information processing systems – computer graphics- graphic kernel system for three dimension (GKS-3D) functional description. Technical Report ISO Document Number 9905:1988(E)"*, American National Standars Institute, NewYork, 1988.

-[GOL94] Goldschen, A. and Garcia, O. and Petajan, E., "*Continuous Optical Automatic Speech Recognition"* Proceedings of the 28th Asilomar Conference on Signals, Systems, and Computers," pp. 572-577. IEEE 1994.

-[GOL97] D.E. Goldberg, *"IMPROV: a System for Real-Time Animation of Behaviour-based Interactive Synthetic Actors"*, Creating Personalities for Synthetic Actors, Trappl R. and Petta P. Eds., Springer-Verlag, pp.58-73. 1997.

-[GOS94] Rich Gossweiler, Robert J. Laferriere, Michael L. Keller, Pausch, *"An Introductory Tutorial for Developing Multiuser Virtual Environments"*, Presence: Teleoperators and Virtual Environments, Vol. 3, No. 4. 1994.

-[GOS96]James Gosling, Bill Joy y Guy Steele.*"The Java Language Specification"*, Addison Wesley, Reading Massachusetts. 1996.

-[GOT99] Taro Goto, Marc Escher, Christian Zanardi, Nadia Magnenat-Thalmann **"***MPEG-4 based animation with face feature tracking".* CAS '99 (Eurographics workshop), Milano, Italy, , Springer, Wien New York, pp.89-98. 1999

-[GOU71] H. Gouraud. *"Continuous shading of curved surfaces"*. IEEE Transactions on Computers, 20(6):623--629. 1971.

-[GOU89] Gourret J.P., Magnenat Thalmann N., Thalmann D., *"Simulation of Object and Human Skin Deformations in a Grasping Task",* Proceedings Siggraph'89, 23, 3, pp. 21-30. 1989.

-[GRA95] J. Granieri, J. Crabtree, and N. Badler. *"Off-line production and real-time playback of human figure motion for 3D virtual environments"*. VRAIS Conf. IEEE Presss. 1995.
-[GRA96] D.M.Gravila and L.S. Davis. *"3D Model-Based Tracking of Humans in Action, A MultiView Approach"*. Proc.CVPR'96. 1996.

-[GUT02] M. Gutierrez, F. Vexo, D. Thalmann. "A mpeg-4 virtual human animation engine for interactive web based applications". Proceedings of the IEEE International Workshop on Robot and Human and Interactive Communication, pp.554–559. Berlin. Septiembre 2002.

-[GUT03] M. Gutierrez, F. Vexo, D. Thalmann. *"Controlling Virtual Humans Using PDAs"*. The 9th International Conference on Multi-Media Modeling (MMM'03), Taiwan. Enero 2003.

-[HAN99] Humanoid Animation Working Group. *"Specification for a Standar Humanoid Version 1.1"*. 1999.

-[HAY97] B. Hayes-Roth, R. Van Gent y D. Huber, *"Acting in Character"*, Creating Personalities for Synthetic Actors,Trappl R. and Petta P. Eds., Springer-Verlag, pp.92-112. 1997.

-[HOD97] Jessica K. Hodgins y Nancy S. Pollard. *"Adapting Simulated Behaviors For New Characters".* Siggraph97 Conference Proceedings. pp. 153-162. 1997.

-[HOP93] Hughes Hoppe, Tony DeRose, Ton DuChamp, John McDonald, and Werner Stuetzle. *"Mesh optimization"*. Siggraph 93 Conference Proocedings (computer Grapichs, Annual Conference Series 1993), páginas 19-26, Agosto 1993.

-[HSU92] Hsu W.M., Hugues J.F., Kaufman H., "*Direct Manipulation of Free-Form Deformations*". Proceedings Siggraph'92, 26, 2, pp. 177-184, 1992.

-[IEE93] Institute of Electrical and Electronics Engineers, International Standar, "*ANSI/IEEE Standard 1278-1993, Starndar for Information Technology, Protocols for Distributed Interactive Simulations*". 1993.

-[ISL01] Isla D., Burke R., Downie M., y Blumberg B. *"A layered brain architecture for synthetic creatures"*. In Proceedings of The International Joint Conference on ArtiÞcial Intelligence. 2001

-[ISO91] *International standard. "Road vehicles- Vehicle dynamics and road-holding ability- Vocabulary"*. International Organization for Standardization . Geneva. 1991. 28 p. ISO 8855:1991.

-[JON94] Erick Jonhson. *"Hight Quality computes graphics in entertainment"*. En Designning Real-Time Graphics for Entertainmente. SIGGRAPH'94 Course Notes. Agosto 1994.

-[JON96] Michael Jones Silicon Graphis Computer Systems. *"Designing Real-Time 3D Graphics for Entertenainment"*. Siggraph'96 Course. 1996.

-[KAL91] P. Kalra, A. Mangili, N.M. Thalmann, D. Thalmann. "*SMILE : A Multilayered Facial Animation"*. System. Proc. of Conference of Modeling in Computer Graphics, Springer, Tokyo. 1991.

-[KAL92] Kalra P., Mangili A., Magnenat-Thalmann N., Thalmann D., *"Simulation of Facial Muscle Actions Based on Rational Free-Form Deformations"*, Computer Graphics Forum, 2, 3, Blackwell Publishers, pp. 65-69. 1992.

-[KIM03] Tae-hoon Kim, Sang Il Park, Sung Yong Shin. *"Rhythmic-Motion Synthesis Based on Motion-Beat Analysis"*. Proceedings of ACM SIGGRAPH´03. pp.392-401. 2003.

-[KIM02] Kim T. Y. *"Modeling, Rendering, and Animating Human Hair"*. PhD thesis, University of Southern California. 2002.

-[KOE97] Koechling, J., Crane, A., & Raibert, M. *"People are Not Tanks: Live Reckoning for Simulated Dismounted Infantry Using DI-Guy,"* Proceedings of the Fall 1997 Simulation Interoperability Workshop, pp. 781-786. 1997.

-[KOG94] Y.Koga, K.Kondo, J. Kuffner, ay, J.C. Latombe. *"Planning motions with intentions"*. ACM Computer Graphcis Annual Conf. Series, pp. 395-408. 1994.

-[KOV02] Kovar, Gleicher, Pighin. *"Motion graphs"*. Proceedings of ACM SIGGRAPH'02. p473-482. 2002.

-**[KRO94]** K.Rohr. Towasrd *"Model-Based Recognition of Human Movementes in Image Secuences"*. CVGIP: Image Understanding, vol 59, no.1 pp. 95-115. 1994

-[MAC94] M. R. Macedonia, M.J Zyda, D.R. Pratt, P.T. Barham, *"NPSNET: A Network Software Architecture for Large-Scale Virtual Environments"*, Presence: Teleoperators and Virtual Environments , Vol.3, No. 4. 1994.

-[MAC96] MacCracken R., Joy K.I.,*"Free-Form Deformations with Lattices of ArbitraryTopology"*, Computer Graphics Research Laboratory, University of California, Technical report CSE-96-7.1996.

-[MAE94] P. Maes, *"Modeling Adaptive Autonomous Agents"*, Artificial Intelligence, vol. 1, no.1/2, pp.135-162. 1994.

-[MAE95] P. Maes, T. Darrell, B. blumbert, y A. Pendland. The ALIVE system: *"Full-body interaction with autónomous agents"*. In N. Magnenat-Thalmann and D. Thalman, editors, Computer Animation, pp.11-19. IEEE Computers Society Press, Los Alamitos, CA. 1995.

-[MAG88] Magnenat Thalmann N., Laperriére R. Thalmann D., *"Joint-Dependent Local Deformations for Hand Animation and Object Grasping"*, Proceedings Graphics Interface'88, pp. 26-33. 1988.

-**[MAG98]** N.Magnenat-Thalmann, P.Kalra, M.Escher, *"Face to Virtual Face"* Proceedings of the IEEE. VOL.86, NO.5. 1998.

-[MAG00] Nadia Magnenat-Thalmann, Sunil Hadap, Prem Kalra, *"State of the Art in Hair Simulation"*, International Workshop on Human Modeling and Animation, Seoul, Korea. 2000.

-[MAH94] S. Mah, T. W. Calvert y W. Havens, *"A constraint-based Reasoning Framework for Behavioural Animation"*, Computer Graphics Forum, vol. 13, no. 5, pp.315-324. 1994.

-[MAT98] "*The Matrix and Quaternions FAQ.version 1.4*" http://www.glue.umd.edu/~rsrodger. Diciembre 1998.

-[MCK90] M. McKenna, S. Pieper y D. Zeltzer, *"Control of a Virtual Actor: The Roach"*, Proc. 1990 Symposium on Interactive 3D Graphics, Computer Graphics, vol. 24, no. 2, , pp.165-174. 1990.

-[MCL91]Patricia McLendon.*"Graphics Library Programming Guide"*. SGI, 1991.

-[MOC96] Laurent Moccozet. *"Hand Modelling and Animation for Virtual Humans"*. PhD Thesis. University of Geneve. 1996.

-[MOL99] Tom Molet, Amaury Aubel, Tolga Çapin, Stéphane Carion , Elwin Lee, Nadia Magnenat-Thalmann, Hansrudi Noser, Igor Pandzic , Gaël Sannier, Daniel Thalmann *"ANYONE FOR TENNIS?"* Presence, Vol. 8, No. 2, MIT press, pp.140-156. 1999.

-[MOH03] Alex Mohr, Michael Gleicher. *"Building Efficient, Accurate Character Skins from Examples"*. Proceedings of ACM SIGGRAPH´03. pp.562-568. 2003.

-[MON91] Monheit, G., And Badler, N.I. *"A kinematic model of human spine and torso"*. IEEE Computer Graphics and Applications. pp. 29-38. 1991.

-[MON97] Montrym J.S., Baum D.R, Dignam D.L y Migdal C.J. *"InfiniteReality: A Real-Time Graphics System"*. Proceedings of SIGGRAPH'97. pp 293-302. 1997.

-[NAG94] K.Nagao and A.Takeuchi, *'Speech Dialogue With Facial Displays: Multimodal Human-Computer Conversation."* Proceedings of the 32nd Annual meetings of the Association for Computational Linguistics, pp.102-109. 1994.

-[NEG95] Nicholas Negroponte. *"Being Digital"*. Random House. 1995.

-[NEI94] Neider,Jackie, Tom Davis and Mason Woo. *"OpenGL Programming Guide"*. Addison-Wesley, Reading, Mass,1994.

-[NG96] H.N NG and R.L.Grimsdale. *"Computer graphic techniques for modeling cloth"*. IEEE Computer Graphics and Applications, 16:28-41. 1996.

-[NIE95] W. Niem, H. Broszio. *"Mapping Texture from Multiple Camera Views onto 3D-Object Models for Computer Animation"*. Proceedings of the International Workshop on Stereoscopic and Three Dimensional Imaginn. Santorini, Grecia.1995

-[NOM97] T. Noma and N. Badler. *"A virtual human presenter"*. IJCAI'97 Worskhop on Animated Interface Agentes, Nagoya, Japan. 1997.

-[NOS96] H. Noser , L.S. Pandzic, T.K. Capin, N. Magnenat Thalmann y D. Thalmann, *"Playing games through the Virtual Life Network"*, Proceedings Artificial Life V, Nara, Japan. 1996.

-[LAI01] John E. Laird. *"It knows what you're going to do: adding anticipation to a quakebot"*. Proceedings of the Fifth International Conference on Autonomous Agents. pp 385–392. 2001.

-[LAM94] Lamousin H. J., Waggenspack W. N., *"NURBS-based Free-Form Deformations"*, IEEE Computer Graphics and Applications, 14, 16, pp.59-65. 1994.

-[LAS00] Joseph Laszlo, Michiel van de Panne, Eugene Fiume. *"Interactive Control For Physically-Based Animation"*. Proceedings of ACM SIGGRAPH'00. pp.201-208. 2000.

-[LAV99] F. Lavagetto, R. Pockaj. *"The Facial Animation Engine: towards a high-level interface for the design of MPEG-4 compliant animated faces" .IEEE Trans. on Circuits and Systems for Video Technology*, Vol. 9, n.2. 1999.

-[LEE99] Jehee Lee, Sung Yong Shin. *"A Hierarchical Approach to Interactive Motion Editing for Human-like Figures"*. Siggraph99 Conference Proceedings. pp. 39-48. 1999.

-[LEE02] Lee J., Chai J., Reitsma P. S. A., Hodgins J. K., y Pollard N. S. *"Interactive control of avatars animated with human motion data"*. Proceedings of ACM SIGGRAPH'02. pp.491-500. 2002.

-[LEW99] Won-Sook Lee, Marc Escher, Gael Sannier, Nadia Magnenat-Thalmann, *"MPEG-4 Compatible Faces from Orthogonal Photos",* Proc. CA99 (International Conference on Computer Animation), Geneva, Switzerland., pp.186-194. 1999.

-[LEW00] J. P. Lewis, Matt Cordner, Nickson Fong. *"Pose Space Deformation: A Unified Approach to Shape Interpolation and Skeleton-Driven Deformation"*. Proceedings of ACM SIGGRAPH'00. pp.165-172. 2000.

-[LEW02] Michael Lewis and Jeffrey Jacobson. ." *SPECIAL ISSUE: Game engines in scientific research". Commun. AC*M, 45(1): pp 27–31. 2002.

-[LIU02] Liu C. K., y Popovic Z. *"Synthesis of complex dynamic character motion from simple animations"*. Proceedings of ACM SIGGRAPH´02. pp.408-416. 2002.

-[LI02] Li Y., Wang T., y Shum H. Y. *"Motion texture: A two-level statistical model for character motion synthesis"*. Proceedings of ACM SIGGRAPH'02. pp .465-472. 2002.

-[LOR93] William E. Lorensen, Willian J. Schoeder, Jonathan A. Zarge. *"Decimation of triangle meshes"*. Siggraph 93 Conference Proceedings (Computer Graphics, Anual Conference Series 1993, pp. 65-70. Agosto 1993.

-[LOZ04] Miguel Lozano, Fran Grimaldo, Fernando Barber *"Integrating miniMin-HSP agents in a dynamic simulation framework"*. 3rd Hellenic Conference on Artificial Intelligence. Springer-Verlag LNAI, Samos (Greece). Mayo 2004.

-[KAL93] P. Kalra. *"An Interactive Multimodal Facial Animation System"*, PhD Thesis nr.1183, EPFL. 1993.

-[KOE97] R. Koenen, F. Pereira, and L. Chiariglione. *"MPEG-4: Context and Objectives", Image Communication Journal, Special Issue on MPEG-4,* Vol. 9, No. 4. 1997.

-**[PAR82]** F.I. Parke. *"A parameterized model for facial animation"*. *IEEE Computer Graphics and Applications*, 2(9): pp. 61-70. 1982.

-[PER95] K. Perlin y A. Goldbeg. *"Real time responsive animation with personality. IEEE Trans. on Visualization and Computer Graphics"*, 1(1) pp. 5-15. 1995.

-[PER96] K. Perlin y A. Goldberg. *"Improv: A system of scripting interactive actors in virtual worlds"*. ACM Computer Graphics, Annual Conf. Series, pp. 205-216. 1996.

-**[PET88]** Petajan, E. D. y Brooke, N. M. and Bischoff, G. J., y Bodoff, D. A. *"An Improved Automatic Lipreading System to Enhance Speech Recognition"* in "Proc. Human Factors in Computing Systems," pp. 19-25. ACM 1988.

-[PET97] Petta P. Trappl R. *"Creating personalities for synthetic actors: Towards autonomous personality agentes"*. Springer-Verlag Series. 1997.

-[PHI88] PHIGS+ Committee, Andries van Dam, chair. *"PHIGS+ functional description, revision 3.0"*. Computer Graphics 22(3). pp. 125-218, July 1988.

-**[PIN00]** A. Pina, F.J. Serón. *"Behaviour modelling, Computer Animation and Ecology"*, *IASTED CGIM 200*0, International Conference on Computer Graphics and Imaging. pp. 313-318. Las Vegas, USA, 2000.

-[POP99] Zoran Popovic, Andrew Witkin. *"Physically Based Motion Transformation"*. Siggraph99 Conference Proceedings. pp. 11-20. 1999.

-[PRA97] David R. Pratt, Shirley M. Pratt, Paul T. Barham, Randall E. Barker, Marianne S. Waldrop, James F. Ehlert, Christopher A. Chrislip, *"Humans in Large-Scale, Networked Virtual Environments"*, *Presenc*e, MIT Press, 6(5), pp. 547-564. 1997.

-[PRE02] M.Preda, F.Preteux. *"Advanced animation framework for virtual character within the MPEG-4 standard"*. Proceedings IEEE International Conference on Image Processing (ICIP'2002), Rochester, NY. Septiembre 2002.

-[REI94] B.Reich, H.Ko, W. Becket, y N.Badler. "Terrain reasoning for human locomotion". Proc. Computer Animation´94, pp. 996-1005. IEEE Computer Society Press. 1994.

-[REY87] C.W. Reynolds, *"Flocks, Herds and Schools: a Distributed Behavioral Model"*, ACM Computer Graphics, vol. 21, no. 4, pp. 25-34. 1987.

-**[RIC99]** J.Rickel, J.Lester, c.Elliot. *"Lifelike pedagogical agenes and affective computing: An exploratory synthesis"*. Artificial Intelligence Today. Springer. 1999.

-[ROH94] Rohlf, J.y Helman J., *"IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics"*. SIGGRAPH Proceedings. 1994.

-[ROS93] Jarek Rossignac and P. Borrel. *"Multi-resolutión 3D approximatión for rendering complex scenes"*. Second Conference on Geometric Modelling in Commputer Graphics. pp. 453-465, Junio 1993.

-[ROS96] C. Rose, B. Guenter, B. Bodonheimer, and M. Cohen. "*Efficiente generation of motion transitions using spacetime constrainsts"*. Siggraph96 Conference Proceedings. pp.147-154. 1996.

-[ROU96] D. Rousseau and B. Hayes-Roth. *"Personality in synthetic agents".* Technical Report KSL-96-21, Stanford Knowledge Systems Laboratory, 1996.

-[RUM96] Rumbaugh, Blaha, Premerlandi, Eddy, Lorensen. *"Modelado y Diseño orientados a objetos"*. Ed. Prentice Hall. 1996.

-[SAE76] "*SAE7618. Vehicle Dynamics Terminology", SAE J670*e, Society of Automotive Engineers, Inc.,Warrendale, PA, 1976.

-[SAN03] Peter Sand, Leonard McMillan, Jovan Popovic. *"Continuous Capture of Skin Deformation"*. Proceedings of ACM SIGGRAPH´03. pp.578-586. 2003.

-[SED86] Sederberg T.W., Parry S.R., *"Free-Form Deformations of Solid Geometric Models"*, Proceedings Siggraph'86, Vol. 20, No 4, pp.151-160. 1986.

-[SEG94] Mark Segal and Kurt Akeley. "*The Design of the OpenGL Graphics Interface"*. Silicon Graphics Computer Systems. 1994.

-[SEO00] Hyewon Seo and Nadia Magnenat-Thalmann. *"LoD Management on Animating Face models"*. Proc. IEEE Virtual Reality 2000, (New Brunswick, USA) IEEE Computer Society Press, pp. 161-168. 2000.

-[SCH97] Ferdi Scheepers, Richard E. Parente, Wayne E. Carlson, Stephen. F.May. *"Anatomy-Based Modelling of the Human Musculature"*. Siggraph97 Conference Proceedings. pp. 163-172.Agosto 1997.

-[SHO85] Shoemake, K.. "*Animating Rotations with Quaternion curves"*. Computer Graphics19(3), 245-54. Proc. SIGGRAPH'85).1985.

-[SHO87] Shoemake, K.. *"Quaternion Calculus and Fast Animations",* SIGGRAPH Course Notes pp. 101-21, 1987.

-[SIL95] Silicon Graphics, Inc. *"The OpenGL Graphics System: A Specification (Version 1.1)".* 1995.

-[SMI97] T. Smith, J. Shi, J. Granieri y N. Badler. Jackmoo: *"A prototype system for natural lenguage avatar control"*. Pacific Graphics. 1997.

-[SNHa96] SNHC Ad Hoc Groups, *"Draft Specification of SNHC Verification Model 1.0"*, Document No. MPEG96/N1364, October 1996 Chicago Meeting of ISO/IEC JTC1/SC29/WG11.

-[SNHb96] SNHC Face/Body Ad Hoc Group, "*Face and Body Definition and Animation Parameters"*, Document No. MPEG96/N1365, Chicago Meeting of ISO/IEC JTC1/SC29/WG11. Octubre 1996.

-[SPR95] R.Sproat y J.Olive, "*An Approach To Text-to-Speech Synthesis."* W.B.Kleijn & K.K.Paliwal (Eds) *Speech Coding and Synthesis*, Elsevier Science. 1995.

-[STA97] S. Stanfsfield, D. Carlson, R. Higtower y A. Sobel. *"A protopype VR system for training medics".* MMVR Proc. 1997.

-[STE92] Jeff Stevenson. *"PEXlib specification and C languaje binding, version 5.1P"*. The X Resource, Special Issue B, September 1992.

-[STR93] Strauss, Paul and Rikk Carey. *"An Object-Oriented 3D Graphics Toolkit"*. Proceedings of SIGGRAPH 93. In Computer Graphics, Annual Conference Series, 341-349. 1993

-[TAN00] Tanco L. M., y Hilton A.. *"Realistic Synthesis of Novel Human Movements from a Database of Motion Capture Examples"*. Proc. the IEEE Workshop on Human Motion. pp.137-142. 2000.

-[TELL92] Seth Teller and Carlo Sequín. Visibility preprocessing for interactive walkthroughs. Computer Graphics (Proceedings SIGGRAPH'92), 26(2):61-69, Agosto 1992.

-[THA95a] D. Thalmann, T. Çapin, N. Magnenat Thalmann, I. Pandzic, *"Participant, User-Guided and Autonomous Actors in the Virtual Life Network VLNET"*, *Proc. ICAT/VRST '9*5 pp. 3-11. Chiba, Japan. 1995.

-[THA95b] Nadia Magnenat Thalman y Prem Dalra. *"The simulation of a virtual TV presenter. Computer Graphics and Applications"* (Proc. Pacific Graphics '95). pp. 9-21. Word Scientific. Agosto 1995.

-[THA96] D. Thalmann, *"A new generation of Synthetic Actors: the Real-Time and Interactive Perceptive Actors"*, Proceedings Pacific Graphics 96, Taipeh, Taiwan. 1996.

-[TER90] Terzopoulos, D. and K. Waters. *"Analysis of facial images using physical and anatomical models"*. *IEEE Proceedings of ICCV conference*, Osaka: Japon. pp. 727-732. 1990.

-[TER99] D. Terzopoulos, *"Artificial life for Computer Graphics"*, Communications of the ACM, vol. 42, no. 8. pp. 33-42. 1999.

-[TES95] A. Tesey and G. L. Foresti and C.S.Regazzoni. *"Human Body Modeling form People Localization and Tracking form Real Image Secuences"*. Image Processing and it´s Applications. Conferenze Publication No. 410. pp. 806-809. 1995.

-[TOL97] Tolga K. Capin, Igor Sunday Pandzic, Nadia Magnenat Thalmann, Daniel Thalmann. "*A Dead-Reckoning Algorith for Virtual Human Figures*" Proc. VRAIS'97 pp. 161-169. 1997.

-[TOL00] D. Tolani, A. Goswami, y N. Badler. *"Real-time inverse kinematics techniques for anthropomorphic limbs"*. Graphical Models 62(5). pp.353-388. Septiembre 2000.

-[TOS96] Naoko Tosa, Ryohei Nakatsu, *"The Esthetics of Artificial Life: Human-Like Communication. Character 'MIC' and Feeling Improvisation Character 'MUSE'"*, Proc. Artificial Life. 1996.

-[TU94] X. Tu y D. Terzopoulos, *"Artificial Fishes: Physics, Locomotion, Perception, Behavior"*, Computer Graphics Proceedings , SIGGRAPH'94, pp. 42-48. 1994.

-[TUR92] Greg Turk. *"Re-tiling polygonal surfaces"*. Computer Graphics (Proceedings SIGGRAPH'92), 26(2). pp. 55-64, Agosto 1992.

-[UDA93] Uda N., Kimura F., Tsuruoka S., Miyake Y., Shunlin S., Tsuda M., *"Expansion of the Solid Modeler's Functions for Use in Designing a Model of the Human Body"*, New Advances in Computer Aided Design & Computer Graphics, (Eds Zesheng Tang), International Academic Publishers, pp. 357-363, 1993.

-[UNU95] Munetoshi Unuma, Ken Anjyo, Ryozo Takeuchi, *"Fourier Principles for Emotion-Based Human Figure Animation"*, Proceedings of ACM SIGGRAPH'95. 1995.

-[UPS90] Steve Upstill. *"The RenderMan Companion"*. Addison-Wesley. Reading, Mass., 1990.

-[VIA92] M.L. Viaud y H. Yahia. *"Facial animation with wrinkles"*. In 3nd Workshop on Animation, *Eurographics '92*, Cambridge. 1992.

-[VOL00] Pascal Volino, Nadia Nadia Magnenat-Thalmann**,** *"Implementing fast Cloth Simulation with Collision Response"***.** Computer Graphics International. 2000.

-[WER94] J. Wernecke, *"The Inventor Mentor:Programming Object-Oriented 3D Graphics with Open Inventor,"* Release 2 Addison-Wesley Publishing Company, ISBN 0-201-62495-8, 1994.

-[WIN97] J.Wingbermuhle, S. Weik, A. Kopernik. *"Highly Realistic Modeling of Persons for 3D VideoConferencing Systems"*. Electronic Proceedings IEEE Workshop on Multimedia Signal Processing. 1997.

-[WU96] Yin Wu, Nadia Magnenat Thalmann and Daniel Thalmann, *"A Dynamic Wrinkle Model in Facial Animation and Skin Ageing"*, JVCA, Vol 6, pp. 195-205. 1995.

-[WAN02] Wang X. C., y Phillips C. *"Multi-weight enveloping: Least-squares approximation techniques for skin animation"*. Proceedings of the ACM SIGGRAPH Symposium on Computer Animation. pp.129-138. 2003.

-[WAT87] K. Waters. *"A muscle model for animating three-dimensional facial expressions". Computer Graphics*, 21(4): pp.17-24. 1987.

-[WAT92] Alan and Mark Watt. "Advanced *Animation and Rendering Techniques: Theory and Practice"*. Addison Wesley.1992.

-[WHI95] While Jones. "*Quaternions quickly transform coordinates without error buildup"*. EDN Magazine. Marzo 1995.

-[YOO00] Yoon S., Burke R., y Blumberg B. *"Interactive training for synthetic characters"*. Proceedings of AAAI 2000.