Desarrollo de técnicas cooperativas para un editor de objetos 3D

Proyecto de Fin de Carrera
Ingeniería Informática

David Sánchez Crespillo
Universitat de les Illes Balears

Año 2000

Índice General

1. Introducción

En esta memoria presentamos un conjunto de módulos y técnicas implementados durante el desarrollo de un editor interactivo de objetos tridimensionales para trabajo cooperativo. Estos módulos y técnicas están destinados a hacer posible el trabajo cooperativo entre múltiples usuarios simultáneos. El editor servirá para que arquitectos e ingenieros puedan trabajar cooperativamente sobre una misma escena tridimensional, aunque se encuentren situados en distintos lugares geográficamente dispersos (incluso en distintos países europeos).

El hecho de que el editor esté enfocado al trabajo cooperativo es uno de los puntos más importantes del proyecto. Es necesario tener una arquitectura de red subyacente y utilizar los protocolos de aplicación adecuados, de manera que el trabajo cooperativo sea posible, razonablemente rápido y fiable.

Del mismo modo, el editor ha de ser aplicado al ámbito de la construcción. Esto significa que debe incorporar una serie de características de edición que son demandadas por los usuarios finales de la aplicación, es decir, arquitectos e ingenieros. Estas características de edición deben funcionar de forma cooperativa.

Por otra parte, el editor implementa funciones que han sido diseñadas e implementadas por distintas empresas y centros de investigación dentro del mismo proyecto. Por ello, su diseño general es abierto y permite integrar estas funciones desde distintas fuentes. Además, ha de ser extensible, de forma que se puedan introducir nuevas características sin afectar al resto de la aplicación.

Otra de las características que presenta el editor es que se trata de una aplicación desarrollada por un equipo de programadores, no por una sola persona. La comunicación entre programadores, la estabilización de código, la implantación de nuevas versiones, presentan problemas que se deben solucionar, para que el tiempo de desarrollo no se alargue y se puedan cumplir los plazos de entrega del producto software.

A lo largo de esta memoria, explicaremos de qué manera hemos implementado el funcionamiento cooperativo de las operaciones básicas de edición en el editor, y el de otras operaciones más avanzadas, como las técnicas de portapapeles y la vuelta atrás.

En la primera parte, explicamos los conceptos generales en los que se ha apoyado el desarrollo de las técnicas cooperativas. El Capítulo 2 explica la estructura general del editor 3D y el sistema general. El Capítulo 3 describe Open Inventor, el sistema de librerías de desarrollo 3D utilizado en la aplicación. Por último, el Capítulo 4 describe las plataformas de soporte al trabajo cooperativo que hemos utilizado para la implementación.

La segunda parte explica las técnicas implementadas y extendidas para hacer posible el trabajo cooperativo con el editor de objetos 3D. En el Capítulo 5 explicamos la forma en que hemos implementado el manejo de los datos tridimensionales en el marco cooperativo. El

Capítulo 6 explica la implementación del control de concurrencia usando exclusión mutua entre usuarios al acceder a objetos 3D, mediante una técnica basada en selecciones de objetos. El Capítulo 7 explica la implementación de las funciones básicas del editor (entrada y salida, manipulación geométrica interactiva, edición de materiales y parámetros de iluminación) desde el punto de vista cooperativo. Por último, los Capítulos 8 y 9 explican la implementación cooperativa de dos funciones más avanzadas: las operaciones de portapapeles y la vuelta atrás.

Se ha incluido una sección de apéndices que reflejan aspectos importantes, tanto del desarrollo de las técnicas cooperativas, como del diseño general de la aplicación. Hemos considerado necesario incluirlos, puesto que reflejan una dimensión complementaria del proceso de implementación. El Apéndice A explica el sistema de desarrollo concurrente adoptado por el equipo de programadores. El Apéndice B, por último, explica las consideraciones seguidas sobre portabilidad y multiplataforma durante el desarrollo de la aplicación.

Este editor ha sido diseñado e implementado dentro del marco del proyecto europeo M3D, que reune a seis socios de tres países europeos, entre ellos la Universitat de les Illes Balears. Los seis socios son los siguientes:

- Universitat de les Illes Balears (Palma de Mallorca, España).
- ADETTI (Lisboa, Portugal).
- European Design Center (Eindhoven, Holanda).
- Oficina de Arquitectura (Lisboa, Portugal).
- IDOM (Barcelona, España).
- ArqMaq (Palma de Mallorca, España).

Tres de los socios (UIB, ADETTI y EDC) actúan como desarrolladores, mientras que los otros tres socios (OA, IDOM y ArqMaq) actúan como usuarios de los productos generados en el proyecto. Estos usuarios aportan sugerencias y líneas de trabajo que determinan algunas de las características implementadas dentro del editor.

Parte I.

Conceptos generales

2. El sistema M3D y el editor

El editor cooperativo de objetos 3D es la aplicación sobre la que hemos implementado las funciones y técnicas explicadas en esta memoria. Este editor recibe el nombre de *M3D Editor* ([SC00b], [SC99a]) y está integrado en el sistema M3D. En este capítulo describimos el sistema M3D, los módulos de que consta, el propósito del editor y su estructura.

2.1. El sistema M3D

El sistema M3D ([Gal97], [Luo98], [Gal99], [Luo99], [Luo00]), implementado y financiado por el proyecto europeo ESPRIT 26287 del mismo nombre, surge a partir de la problemática a la hora de integrar y coordinar diseños de distintas especialidades en una producción arquitectónica. En el diseño y construcción de un edificio es necesario que especialistas diferentes trabajen coordinadamente. Sin embargo, cuando se han de integrar los planos generados por un arquitecto con los diseños de estructuras y con los diseños de cañerías de otros dos ingenieros distintos, es fácil que aparezcan problemas de coherencia. Esto es debido a la falta de un soporte tecnológico integrado que facilite el intercambio de información. Por ello, habitualmente tiene lugar un ciclo de "prueba y error" hasta que se obtiene el resultado final. Este problema ralentiza el proceso de producción, ya que los problemas de coherencia son detectados normalmente en fases avanzadas del proyecto, incluso en la fase de construcción, lo que resulta extremadamente costoso de solventar.

El proyecto M3D presenta una solución doble a este problema. Las dos partes de esta solución son las siguientes:

- Por un lado, el proyecto propone un replanteamiento del proceso de negocio, basado en la utilización de herramientas tecnológicas y de comunicación.
- Por otra parte, el proyecto presenta un sistema, el sistema M3D, que dará soporte al trabajo cooperativo básico en la producción arquitectónica, durante la fase de diseño.

El objetivo técnico principal del sistema M3D consiste en hacer posible el acceso y el uso de la información técnica (principalmente información geométrica) para especialidades distintas (arquitectos e ingenieros). Esto implica que el sistema ha de tener las siguientes capacidades:

- Visualización distribuida de la información geométrica y técnica.
- Modificación interactiva y cooperativa de los objetos CAD por un equipo de diseño arquitectónico.

- Almacenamiento y recuperación de los datos en una base de datos compartida ([SC99b]).

Los objetivos específicos del sistema M3D, que han determinado el diseño posterior de sus elementos, son los siguientes:

- Proporcionar información actualizada lo antes posible a todos los usuarios que participen en un proyecto arquitectónico (arquitectos e ingenieros). Esto evitará que los usuarios desarrollen su trabajo con información desfasada.
- Proporcionar la capacidad de visualización y edición para la agregación y desagregación del diseño principal con el resto de proyectos de especialidades distintas. Esta capacidad de visualización y edición debe darse tanto en modo de trabajo cooperativo como en modo de trabajo monousuario.
- Proporcionar detección básica de inconsistencias durante la agregación de proyectos de diferentes especialidades.
- Identificar las modificaciones en la geometría, así como los autores de éstas, de forma que el coordinador del proyecto pueda contactar con los autores y discutir las modificaciones.
- Almacenamiento de la información complementaria acerca del proyecto arquitectónico, tal como tablas, descripciones textuales y anotaciones.
- Inclusión de herramientas complementarias de comunicación que faciliten la interacción durante el trabajo cooperativo, tales como aplicaciones de audio y videoconferencia.
- Disponibilidad del sistema para *hardware* de bajo coste (estaciones de trabajo basadas en arquitecturas PC, además de las arquitecturas Silicon Graphics).
- Bajo consumo de ancho de banda, que permita utilizar sistemas de comunicaciones de bajo coste. El estándar de conectividad adoptado ha sido la Red Digital de Servicios Integrados (RDSI).

Uno de los ámbitos de aplicación más importantes del sistema M3D tiene que ver con los conceptos de *agregación* y *desagregación* de la información involucrada en un proyecto arquitectónico. La agregación de la información es necesaria para poder comprobar la coherencia de los datos procedentes de las diferentes especialidades. La desagregación de la información, por otro lado, es necesaria para que los distintos especialistas puedan trabajar por separado en sus diseños respectivos. De hecho, el proceso actual de producción arquitectónica consiste básicamente en un ciclo iterativo de agregación y desagregación de las distintas especialidades, hasta obtener el resultado integrado final.

2.1.1. Estructura y aplicaciones

El sistema M3D proporciona un entorno totalmente distribuido destinado al diseño arquitectónico, que consta de las siguientes aplicaciones:

- Un editor cooperativo de objetos tridimensionales: *M3D Editor*.
- Un módulo de verificación de diseños arquitectónicos y detección de interferencias: *M3D Interdetect*.
- Una base de datos compartida especializada en tratamiento de objetos tridimensionales: *M3D Database*.
- Un sistema de soporte al trabajo cooperativo: *M3D Metaconference*.

Como se puede ver en la Figura 2.1, el sistema M3D sigue una estructura de replicación de aplicaciones, de forma que cada una de las estaciones de trabajo ejecuta una instancia del *software* de aplicación. La comunicación tiene lugar mediante el envío de mensajes, y la compartición de datos se realiza mediante el acceso concurrente a la base de datos.

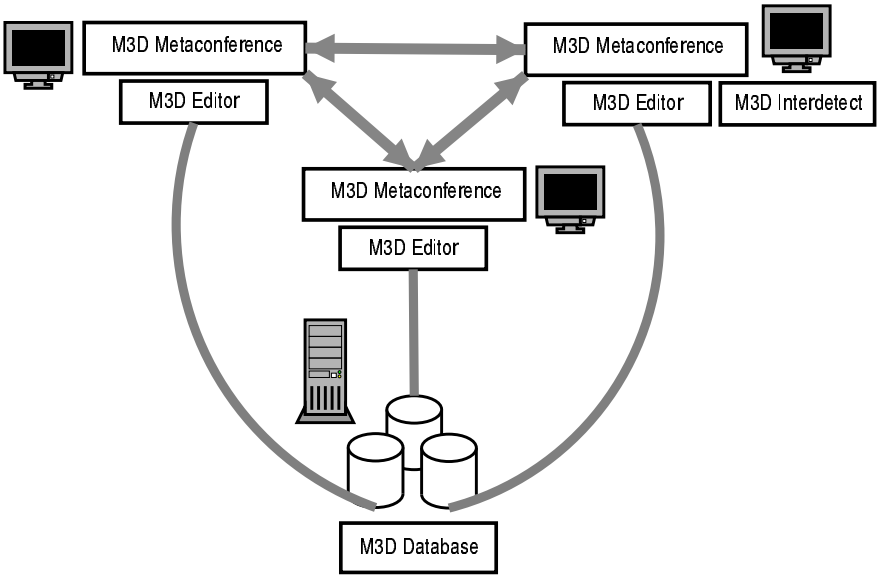

Figura 2.1.: Esquema de la estructura de ejecución del Sistema M3D.

La herramienta principal del sistema M3D es el M3D Editor. Esta aplicación representa el marco principal donde el usuario podrá visualizar, verificar y editar la geometría tridimensional de un diseño arquitectónico. El editor integra, asimismo, otros módulos, tales como M3D Interdetect y los módulos de acceso a la base de datos. Estos módulos serán visibles al usuario mediante su interfaz con el editor.

La base de datos compartida ([SC99b]) contiene las distintas versiones de cada uno de los diseños arquitectónicos y permite la recuperación por contenido de cada uno de los objetos. La recuperación y almacenamiento de los objetos tiene lugar mediante una interfaz HTTP y mediante el módulo de acceso integrado en el editor.

La ejecución de cada una de las aplicaciones es gestionada por una aplicación de alto nivel que controla las funciones relacionadas con el trabajo en grupo. Esta aplicación, llamada

M3D Metaconference ([Alm95]), se encarga de simular un entorno de conectividad total entre los usuarios mediante el control sobre la sesión colaborativa.

2.2. El editor de objetos 3D

El editor se ejecuta de forma distribuida en cada una de las estaciones de trabajo. Cada una de ellas ejecuta una réplica de la aplicación de forma sincronizada. El porqué de este modo de ejecución se ilustra en el Capítulo 4. Un esquema de este modo de ejecución se puede ver en la Figura 2.2. Es posible también la ejecución del programa en modo local.

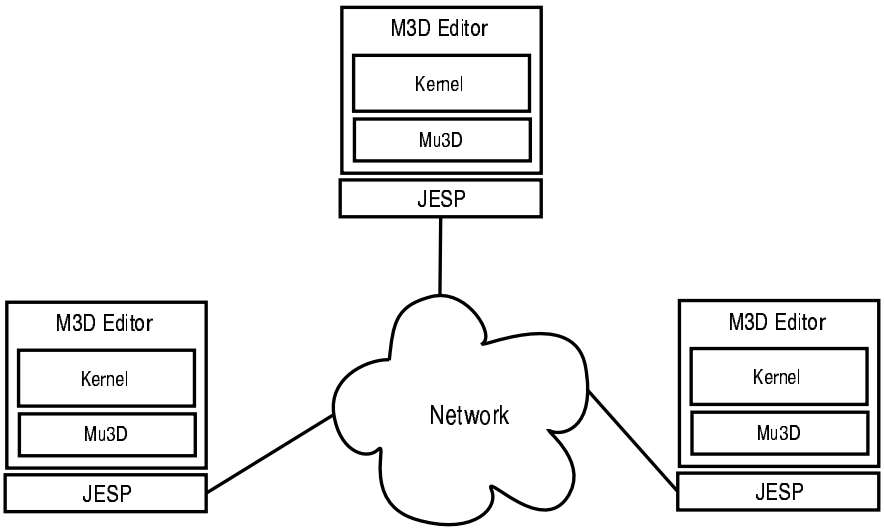

Figura 2.2.: Esquema de la ejecución distribuida del editor

El editor permite realizar operaciones de visualización y edición de los datos tridimensionales que forman un diseño arquitectónico. Además, proporciona un entorno cooperativo de trabajo, en el cual los usuarios pueden observar la localización de cada participante en una sesión cooperativa y el punto de vista de cada uno.

El entorno cooperativo de trabajo del editor sigue diversos estándares de realidad virtual ampliamente aceptados ([Bro96b], [Bro96a], [Bro97a], [Bro97b]), como son los siguientes:

- La inmersión en la escena tridimensional. Los usuarios tienen la capacidad de navegar dentro de la escena y de introducirse virtualmente “dentro” de los edificios o construcciones arquitectónicas.

- La representación de los demás usuarios en la escena. Para ello, se sigue el método de representación mediante *avatares*. Un avatar consiste en la representación de un usuario (generalmente en forma humanoide), con la misma posición y orientación que el usuario tiene en ese momento. De este momento, cada usuario sabe qué usuarios se encuentran en la escena, así como la posición y la orientación de éstos.

Además de la posibilidad de navegar por la escena y de visualizar a los otros participantes, los usuarios pueden modificar los objetos que la forman. Las modificaciones son visualizadas interactivamente en todas las réplicas del editor.

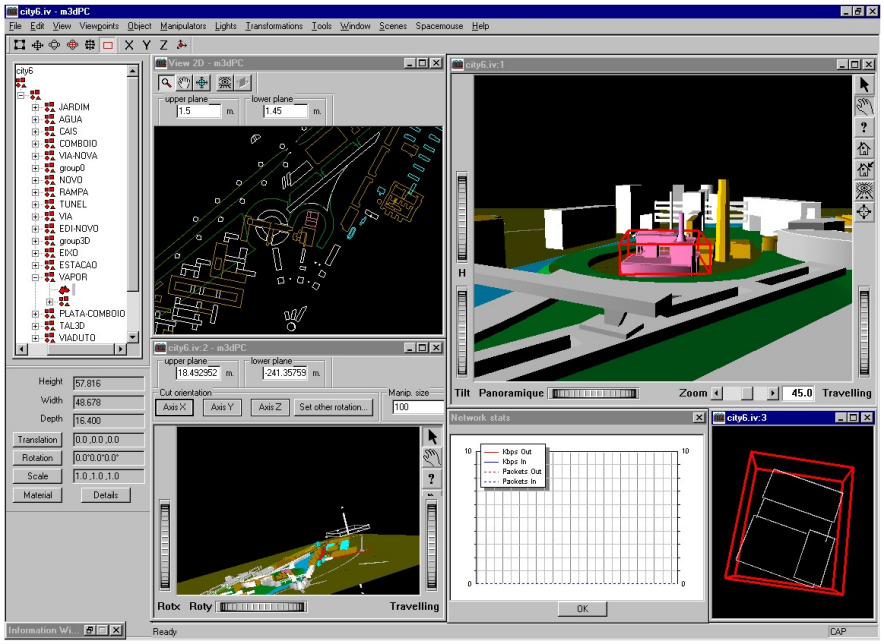

Figura 2.3.: Pantalla principal del editor.

La interfaz de usuario del editor ofrece todas las facilidades posibles para acceder a los datos arquitectónicos de diversas formas. Una imagen de esta interfaz de usuario puede verse en la Figura 2.3.

El editor utiliza el formato VRML de ficheros de datos tridimensionales. Este formato es considerado un estándar para la mayoría de aplicaciones de manejo de objetos 3D. Sobre este formato hablaremos en el Capítulo 3.

2.3. Estructura de la aplicación

Como podemos ver en la Figura 2.4, el editor ha sido diseñado en base a una estructura modular. En la figura podemos ver los distintos módulos de que está compuesto, que explicamos a continuación:

- Módulo de interfaz de usuario (*User Interface*), que implementa las funciones (dependientes del sistema de ventanas sobre el que el editor está implantado) relativas a la gestión de los elementos de interfaz gráfica de usuario.
- Módulo de entrada y salida (*Input and Output)*, que recoge las funciones y estructuras de datos que dan soporte a las rutinas de entrada y salida, tanto a disco como a base de datos.

- Módulo de edición (*Editing*), que agrupa las operaciones realizadas sobre los datos geométricos de una escena.
- Módulo de visualización (*Viewing*), en el cual se implementan las operaciones relativas a la presentación de los datos geométricos de diversas formas útiles para el usuario final.
- Módulo de gestión local de datos (*Local Data Management*), dentro del cual se encuentran las operaciones de gestión y actualización de las estructuras de datos internas.
- Módulo de consistencia de memoria (*Memory Consistency*), que incluye las operaciones necesarias para mantener consistentes las distintas réplicas de los datos geométricos.
- Módulo de soporte al trabajo cooperativo (*CSCW Support*), en el cual se incluye la interfaz con la plataforma de soporte al trabajo cooperativo, implementada en forma de Interfaz de Programación de Aplicaciones (*Application Programming Interface*, en adelante API).

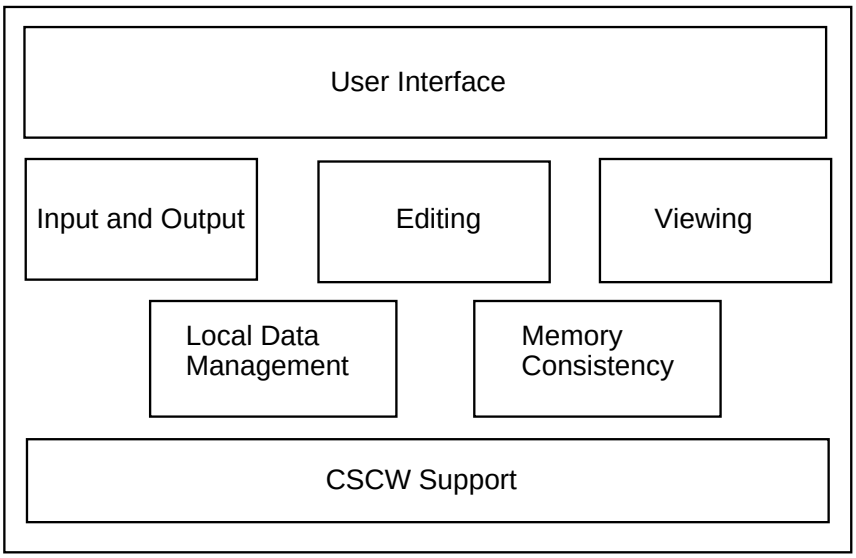

Figura 2.4.: Estructura modular del editor.

Los módulos del editor se comunican entre sí mediante interfaces comunes. La implementación de la mayoría de ellos se realiza siguiendo el paradigma de la programación orientada a objetos ([Bud94]), mediante clases. El desarrollo del editor ha sido realizado íntegramente en lenguaje C++ ([HO93], [Lip91]). Las operaciones de alto nivel de manejo de los objetos tridimensionales han sido implementadas utilizando las librerías estándar Open Inventor, que explicamos con más detalle en el Capítulo 3.

3. Open Inventor y VRML 97

En el desarrollo de cualquier aplicación 3D con una mínima envergadura, es necesario utilizar funciones y estructuras de datos de alto nivel, que liberen al programador de la complejidad que supone desarrollar algoritmos de manejo de las estructuras de datos, de visualización, o de gestión del hardware de visualización de gráficos.

El conjunto de librerías Open Inventor ([Wer94a], [Wer94b]), desarrollado por la empresa Silicon Graphics[1], presenta una interfaz de programación de aplicaciones que permite al programador construir una aplicación utilizando pocas instrucciones de bajo nivel. Open Inventor está construido a su vez aprovechando las librerías OpenGL ([SGI95]), desarrolladas también por Silicon Graphics, que constituyen un estándar *de facto*.

Open Inventor consiste, desde el punto de vista del programador, en un conjunto de clases en lenguaje C++ que permiten diseñar e implementar aplicaciones que sigan el paradigma de la programación orientada a objetos. Las relaciones de herencia entre las distintas clases permiten definir una jerarquía, de forma que el programador pueda escoger las que más se adapten a sus necesidades. Además, Open Inventor define un formato de fichero que permite intercambiar la información generada por distintas aplicaciones rápidamente, sin necesidad de implementar algoritmos de conversión a otros formatos.

A lo largo de todo este capítulo nos centraremos especialmente en los aspectos de Open Inventor que hemos utilizado en la aplicación que nos ocupa. Realizar una introducción que cubra cada aspecto de estas librerías ocuparía mucho espacio y está fuera del alcance de esta memoria. Por eso, cubriremos en ella únicamente los aspectos más básicos y los aspectos avanzados que son utilizados en la aplicación.

En primer lugar presentaremos la estructura de datos fundamental utilizada por estas librerías, el grafo de escena, y la manera de manipularlo. A continuación, describiremos las operaciones más importantes en una escena tridimensional, en términos de recorridos sobre el grafo y de modificaciones sobre éste. A continuación, introduciremos algunos de los nodos del grafo más importantes y sus propiedades. Finalmente, veremos de qué manera se visualiza la escena asociada a un grafo de escena y cómo se relaciona con el sistema operativo que le da soporte.

El formato VRML 97, propuesto por el mismo equipo que diseñó Open Inventor, es el formato básico adoptado para el proyecto M3D. En la sección final de este capítulo describiremos brevemente su filosofía y funcionamiento, así como las semejanzas y diferencias que presen-

[1] Hace algunos años, Silicon Graphics vendió una licencia de desarrollo sobre Open Inventor a la empresa Template Graphics Software (TGS), que ha desarrollado las últimas versiones de estas librerías para múltiples plataformas. Las versiones con las que se ha trabajado en este proyecto han sido la 2.5.2 de TGS para el sistema operativo Windows NT, y la 2.1.2 de SGI para el sistema operativo Irix.

ta con el formato de fichero de Open Inventor y el soporte que ofrece Open Inventor para el formato VRML 97. También comentaremos los problemas encontrados en este soporte.

3.1. El grafo de escena

La estructura de datos básica en Open Inventor es el *grafo de escena*. Todos los datos de la escena tridimensional, ya sean datos geométricos, matrices de transformación, información sobre el material de cada objeto, o datos de iluminación, están organizados de forma jerárquica. Esta organización jerárquica adquiere forma de *grafo dirigido acíclico* o *Directed Acyclic Graph*, DAG ([Aho88]), de forma que cada nodo del grafo contiene una parte de la información de la escena.

Para operar con un grafo dirigido acíclico se pueden adoptar dos puntos de vista: uno de ellos consiste en tratarlo como un grafo con ciertas restricciones, mientras que el otro consiste en tratarlo como un árbol con algunas extensiones. Las librerías Open Inventor adoptan el segundo punto de vista, de forma que el grafo de escena puede ser manejado de acuerdo con la siguiente definición:

> *Un grafo de escena es un árbol cuyos nodos pueden tener cero o más descendientes, y uno o más ascendientes. En ningún caso un nodo puede ser ascendiente de un ascendiente suyo. Es decir, no se pueden crear ciclos de precedencia.*

Para localizar la posición de un nodo en un grafo de escena es necesario definir qué ascendientes tiene, es decir, qué nodo es el padre, de qué nodo es hijo el padre, etc., hasta llegar a la raíz del árbol. Open Inventor define el concepto de *ruta* (*Path*) para indicar la posición de este nodo como una cadena de índices que comienza en la raíz del árbol y acaba en el nodo especificado.

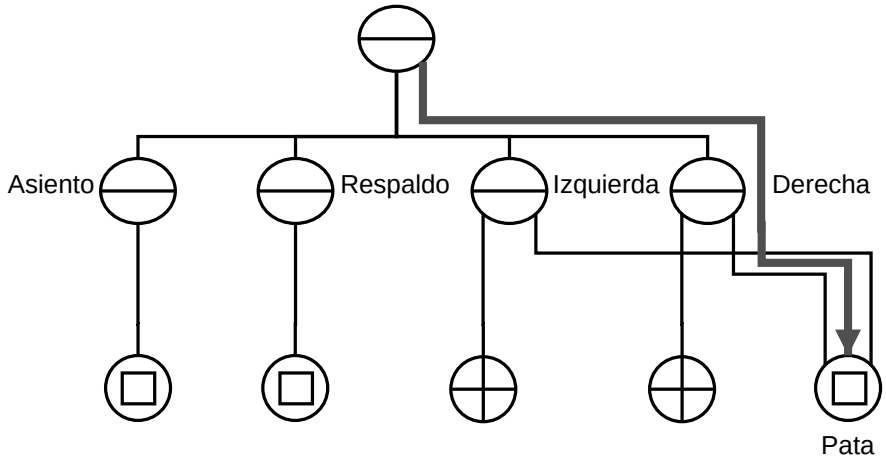

Figura 3.1.: Esquema de un grafo de escena de Open Inventor que reutiliza un nodo.

Para ilustrar la utilidad del concepto de ruta, emplearemos un caso práctico. Supongamos que estamos modelando de forma simplificada el banco de un parque, compuesto de un asiento,

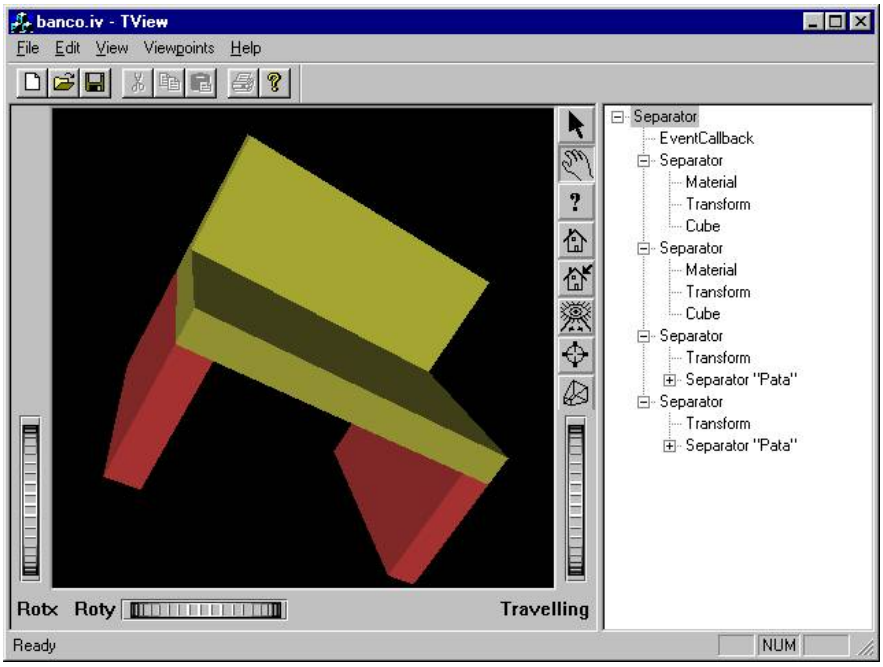

Figura 3.2.: Visualización de un fichero Open Inventor que reutiliza nodos geométricos.

un respaldo y dos patas. Existe la posibilidad de que definamos una sola vez una de las patas y la añadamos como hija dos veces a distintos nodos de agrupación. Cuando el grafo sea recorrido, la acción de redibujado pasará dos veces por la misma pata y la dibujará dos veces. Un esquema de este grafo de escena se puede ver en la Figura 3.1; el resultado de la visualización se muestra en la Figura 3.2.

Al seleccionar una de las dos patas, la diferencia entre una y otra estará en la ruta por la que haya sido seleccionada, ya que se trata del mismo nodo. Así, accederemos a la pata izquierda o a la pata derecha a través de sus rutas correspondientes. Cuando el usuario seleccione un objeto, la ruta adecuada será devuelta por la aplicación. En la Figura 3.1 podemos ver resaltada la ruta a la pata derecha, cuyos índices formarán la cadena 3-1 (la numeración de los índices del grafo comienza en el 0). La ruta a la pata izquierda estaría formada por los índices 2-1.

3.2. Operaciones sobre el grafo de escena

Como toda estructura de datos, el grafo de escena puede ser consultado y manipulado. En esta sección describiremos las operaciones de consulta que pueden ser realizadas sobre un grafo de escena de Open Inventor, así como los cambios que éstas realizan sobre el estado global de una escena. Posteriormente veremos la manera de modificar la estructura de un grafo de escena. Finalmente, describiremos la forma de eliminar nodos del grafo.

3.2.1. Acciones

Para realizar operaciones que consulten el grafo de escena, es necesario efectuar un recorrido en preorden sobre él o sobre un subconjunto. Una operación sobre el grafo de escena recibe el nombre de *acción* en Open Inventor. Una acción puede consistir en visualizar la escena, buscar un nodo, o calcular la *bounding box* asociada a la geometría, entre otras.

Una acción puede ser aplicada a un nodo o a una ruta. El protocolo de utilización de las acciones normalmente se compone de tres fases:

- *Inicialización de la acción*. Asignación de los valores a los parámetros asociados a la acción. Para cada tipo de acción existen distintos métodos específicos de inicialización.
- *Aplicación de la acción*. La aplicación de una acción a un nodo o a una ruta tiene lugar mediante la invocación del método *apply()*.
- *Recogida de los resultados de la acción*. Normalmente, todas las acciones retornan como resultado una ruta o una lista de rutas. Para obtener los resultados se puede invocar a los métodos *getPath()* o *getPaths()*. En caso de retornar otros tipos, se pueden invocar métodos específicos para cada tipo de acción.

Las acciones constituyen la forma estándar de extraer datos de la escena. Cada nodo tiene asociado un comportamiento propio para cada tipo de acción. Los tipos de acciones más importantes en Open Inventor son las siguientes:

- Acción de visualización (*render*). Recorre el grafo asociado y dibuja la geometría que contiene, junto con sus propiedades, en un dispositivo de salida. Esta acción es básica, ya que sin ella no es posible visualizar la geometría de la escena.
- Acción de búsqueda (*search*). Busca un nodo o conjunto de nodos en el grafo de acuerdo con algún criterio, como el tipo o el nombre. Según el tipo de búsqueda, puede retornar el primer nodo, el último, o bien todos los que cumplan con el criterio de búsqueda. Las acciones de búsqueda sirven también para obtener la ruta asociada a un nodo determinado.
- Acción de cálculo de la *bounding box*[2]. Retorna la *bounding box* asociada al nodo o ruta al cual se aplica la acción. Esta acción es útil para diversas aplicaciones, como el cálculo de colisiones entre objetos, o para resaltar un objeto seleccionado dibujando su *bounding box*.
- Acción de escritura (*write*). Realiza la escritura en un fichero del grafo de escena al que se aplica la acción. Este es el método estándar para implementar el almacenamiento en disco de una escena.

[2] Una *bounding box* es el menor prisma cuadrangular recto que incluye completamente el volumen de la geometría asociada.

3.2.2. Estado global de una escena

Open Inventor mantiene un conjunto de *variables globales de estado*; estas variables de estado afectan a la visualización de cada uno de los nodos que son recorrido por una acción de *rendering*. El estado global en un momento dado incluye la matriz de transformación geométrica, el material, el modelo de iluminación y otras características de la escena. Según el tipo de nodo, el estado global puede ser modificado o no en el momento en que una acción de visualización lo recorre.

Las variables globales de estado de Open Inventor tienen relación directa con las variables globales que mantiene OpenGL ([SGI95]). La diferencia radica en que con Open Inventor el programador no necesita asignar valores a las variables de estado explícitamente. Esta operación es realizada de forma implícita por la acción de visualización al recorrer el grafo. El trabajo del programador, pues, consiste en organizar el árbol de forma apropiada, de manera que el resultado de la visualización sea correcto.

3.2.3. Modificaciones del grafo de escena

Para editar una escena tridimensional en Open Inventor, es necesario realizar modificaciones en el grafo de escena asociado. Por ejemplo, para cambiar la posición u orientación de un objeto es necesario modificar el nodo que representa su matriz de transformación.

Existen dos tipos de modificaciones básicas que se pueden realizar sobre un grafo de escena. El primer tipo consiste en modificar un solo nodo de forma aislada. El segundo tipo consiste en modificar la forma en que los nodos están organizados, es decir, cambiar la topología del grafo. Veremos a continuación en qué consiste cada uno.

- Cada uno de los nodos de Open Inventor contiene una serie de *campos*, a los que se asigna un valor. Es posible obtener el valor de estos campos y modificarlo, mediante métodos específicos. La forma estándar de realizar estas operaciones es llamando a los métodos *getValue()* y *setValue()*. De esta forma, se puede modificar cualquier nodo de la escena de forma aislada, cambiando el valor de alguno de sus campos.

- Para modificar la topología del grafo, Open Inventor define operaciones estándar de manejo de árboles, similares a las descritas en [Aho88]. En un grafo de escena existen nodos que funcionan como ascendientes, o *padres* de otros, como veremos con detalle más adelante. Cualquier modificación en la topología del grafo consistirá en añadir, insertar, sustituir o eliminar hijos de uno de estos nodos. Para ello, Open Inventor proporciona distintos métodos estándar, tales como *getNumChildren*, *addChild*, *insertChild*, *replaceChild* y *removeChild*, entre otros.

Una vez modificado el grafo de escena, una aplicación Open Inventor puede detectar el cambio y, si es necesario, aplicar las acciones apropiadas, como por ejemplo, el redibujado de la escena. De esta forma, el programador no tiene que realizar explícitamente llamadas de redibujado. También es posible configurar la aplicación para que sólo se redibuje cuando el programa lo solicite de forma explícita, a efectos de eficiencia.

3.2.4. Borrado de nodos

Open Inventor gestiona la memoria utilizada por los nodos de un grafo de escena de forma que es posible instanciar nuevos nodos explícitamente[3], pero no borrarlos. El borrado de los nodos tiene lugar de forma implícita siguiendo un sistema de referencias. Explicamos a continuación en qué consiste este sistema de referencias y la forma en que afecta al borrado de los nodos.

Cada nodo de Open Inventor contiene una variable entera que indica las veces que ha sido referenciado este nodo. El numero de referencias (llamado en Open Inventor *reference count*) se incrementa cada vez que el nodo es añadido como hijo a un nodo de agrupación[4], y se decrementa cada vez que uno de los ascendientes directos del nodo lo retira (por medio de una llamada al método *removeChild*).

La eliminación de un nodo de Open Inventor de la memoria tiene lugar automáticamente en el momento que el número de referencias pasa de 1 a 0. Es decir, en el momento en que un nodo pasa a no ser hijo de ningún otro, la memoria que ocupaba es liberada.

Este sistema de referencias debe ser gestionado con un cuidado especial en el caso de los nodos raíz de la escena. Como los nodos raíz no tienen ascendientes, su número de referencias es inicialmente cero. Por otro lado, cada vez que una acción es aplicada a un nodo, el comportamiento estándar de ésta consiste en referenciar el nodo al principio,y decrementar el número de referencias al final. Ello implica que un nodo raíz puede hacer que su número de referencias pase de 1 a 0 al finalizar la acción, de forma que el nodo sea borrado. Por ello, es necesario realizar una primera referencia explícita al nodo raíz, con una llamada al método *ref()*.

3.3. Nodos principales en el grafo de escena

Un nodo de Open Inventor encapsula una parte de los elementos tridimensionales que representa el grafo de escena. Existen distintos tipos de nodos, cada uno de ellos representado por una clase C++, con propiedades distintas y reacciones diferentes ante una acción. Veremos a continuación algunos de los tipos de nodos más importantes que podemos encontrar en un grafo de escena de Open Inventor.

3.3.1. Nodos de agrupación

En una estructura jerárquica, como el árbol de escena, algunos de los nodos deben actuar como "padres" de otros. En el caso de Open Inventor, estos nodos son llamados nodos de agrupación, o *grupos*. Existen muchas clases de grupos y cada uno presenta un comportamiento ligeramente distinto ante el recorrido de una acción. Nosotros nos centraremos en los tres tipos más usados: los *grupos* básicos, los *separadores* y los *nodos de selección.*

[3] Mediante el operador *new* de C++.

[4] Como hemos comentado anteriormente, un nodo puede tener más de un padre, siempre que el grafo se mantenga sin ciclos.

Grupos básicos

Los *grupos básicos* agrupan simplemente los nodos que son hijos suyos, sin realizar ninguna otra operación. Es decir, si una acción de visualización llega a un grupo, lo único que hace es recorrer de izquierda a derecha todos los nodos que sean hijos de ese grupo.

La función de este tipo de nodos de agrupación consiste únicamente en propagar la acción a sus hijos, sin modificar ni guardar el estado global de la escena. Su utilidad queda limitada meramente a estructurar la escena tridimensional. Como veremos a continuación, existen otros tipos de grupos más útiles, que ofrecen funcionalidades adicionales.

Separadores

Los *separadores* añaden una funcionalidad más a los grupos básicos: la capacidad de mantener un *estado local*. El estado global de Open Inventor, introducido en la Sección 3.2, es modificado por el separador al ser recorrido por una acción, de la siguiente forma:

- Guardar el estado global de la escena.
- Hacer que la acción recorra todos los nodos hijos del separador.
- Restaurar el estado previamente guardado.

Es decir, las modificaciones en el estado global que se hayan realizado en nodos hijos del separador no tendrán ningún efecto sobre nodos que no sean descendientes de éste, ya que el estado global anterior es restaurado en el momento en que la acción abandona el separador.

Para guardar el estado global, Open Inventor utiliza operaciones de alto nivel proporcionadas por la librería OpenGL. Esta librería mantiene una *pila* en la que se puede almacenar el estado global en distintos momentos de la ejecución del programa. De esta manera, mediante sucesivas llamadas a las operaciones *glPush* y *glPop*, es posible mantener un conjunto de elementos tridimensionales, cada uno con un sistema de coordenadas propio ([SGI95]).

A continuación mostraremos un ejemplo de la utilidad de este tipo de nodo. Supongamos que debemos representar un sistema planetario, con distintos planetas que giran alrededor de una estrella, cada uno con algunos satélites. Utilizando separadores, podemos representar el movimiento de la órbita de los satélites alrededor de cada planeta de forma local, sin preocuparnos del tipo de órbita que describirá el planeta alrededor de la estrella. Incluso es posible modificar el movimiento del sistema completo añadiendo una transformación, sin que ello implique modificar en absoluto el movimiento local de cada elemento del sistema.

Nodos de selección

Un nodo de selección realiza, además de todas las funciones de un separador, otras que facilitan la interacción con el usuario. Estas funciones son las siguientes:

- Cuando el usuario coloca el puntero del ratón sobre uno de los objetos de la escena y hace clic sobre él, la aplicación Open Inventor pone en marcha una función que calcula qué objeto ha sido apuntado, a partir de las coordenadas del ratón. Si el objeto es un

descendiente de un nodo de selección, éste se encarga de gestionar esta función, de forma transparente al programador, y de mantener una lista de objetos seleccionados, de acuerdo con distintas políticas de selección. De esta manera, si el usuario realiza una operación sobre el objeto seleccionado, el programa sólo tiene que consultar al nodo de selección correspondiente, para saber qué nodo está seleccionado.

- Para acceder a la lista de objetos seleccionados, el programador realiza llamadas a métodos propios del nodo de selección. Estos métodos permiten un amplio control de los nodos seleccionados, así como de la selección de nodos nuevos, o retirar nodos de la lista.

- Las políticas de selección tienen que ver con la interacción con el usuario que se desea ofrecer con la aplicación. Según la política que defina el programador de la aplicación, el usuario podrá tener seleccionado uno o varios objetos a la vez, con distintas interfaces posibles.

3.3.2. Nodos de geometría

Los nodos de geometría describen la información geométrica de la escena. Esta información puede venir dada en forma de *primitivas* (Open Inventor define conos, cubos, cilindros y esferas como objetos primitivos), de *representación mediante vértices*, o de *superficies NURBS*. Para más información sobre estos tipos de representación de sólidos, se pueden consultar [Fol92] y [Hea94].

En este apartado nos centraremos principalmente en los nodos que definen una geometría mediante vértices, ya que son los que hemos utilizado más extensamente en la aplicación. Las primitivas definidas por Open Inventor son demasiado simples para ser utilizadas, mientras que las superficies NURBS exigen un tiempo de computación muy elevado, y además no tienen utilidad para la mayoría de proyectos arquitectónicos.

Todos los nodos de geometría que implementan formas mediante vértices en Open Inventor están implementados como clases C++ derivadas de la clase *SoVertexShape*. Existen diversas clases derivadas de esta clase básica, cada una con diferentes particularidades. Dado que comentar cada una de ellas por separado sería demasiado extenso para el propósito de este apartado, detallaremos a continuación algunas de las características comunes que comparten.

La representación de formas geométricas tridimensionales basadas en vértices sigue el modelo llamado *de fronteras* (*Boundary REPresentation*, o BREP). La información almacenada en este modelo se agrupa en dos partes. Por un lado, se guardan todos los vértices del sólido, en un vector global. Por otro lado, se almacena la descripción de todas las caras del sólido, en forma de otro vector de índices ordenados al vector de vértices.

Open Inventor representa este modelo de fronteras mediante nodos del mismo árbol, como hace con todas las demás características de una escena. Así, el conjunto de vértices tridimensionales es representado por un nodo de tipo *SoCoordinate3*, en forma de vector global. Las características del sólido (sentido de ordenación de los vértices en una cara, por ejemplo) son representadas por un nodo de tipo *SoShapeHints*. Otro nodo, de tipo *SoNormal* indica el conjunto de vectores normales a cada cara. Finalmente, otro nodo (*SoFaceSet* o *SoIndexedFaceSet*)

contendrá el vector de índices ordenados respecto al vector global de puntos.

3.3.3. Nodos de propiedad

Los nodos de propiedad modifican el estado global de la escena de una forma particular. Los dos tipos de nodos de propiedad más importantes son los *nodos de material* y las *transformaciones geométricas*.

Los *nodos de material*, como todos los nodos de propiedad excepto las transformaciones, sustituyen los parámetros de visualización globales por los propios. Estos parámetros de visualización pueden ser el color ambiental, el color difuso, el color especular, la transparencia, y otros.

Las *transformaciones geométricas* representan una operación geométrica realizada sobre el resto de la escena que queda por ser recorrida por una acción. Cuando una acción recorre un nodo de transformación, la transformación geométrica representada por este nodo se *concatena* con la transformación geométrica global de la escena. Este es el único tipo de nodo de propiedad que no sustituye completamente el estado global de la escena. Hay muchos tipos de nodos de transformación, cada uno con distintas propiedades, que sirven para definir traslaciones, rotaciones, escalados, etc.

Los nodos de propiedad modifican el estado global del árbol de escena, por lo que deberán ser colocados de manera que los nodos a los que afecten queden siempre a la derecha y/o por debajo. De este modo, cuando una acción cualquiera recorra el árbol de escena, encontrará en primer lugar el nodo de propiedad, y a continuación el resto de nodos a los que esta propiedad se aplica.

3.4. Visualización de una escena

Para visualizar una escena tridimensional o parte de ella, Open Inventor utiliza los llamados *visores* (*viewers*), que se asocian a un grafo o subgrafo de escena. Un visor asociado a un grafo de escena ejecuta una acción de visualización cada vez que parte del grafo es modificada. Cada visor contiene un área de visualización (*Render Area*) asociada, sobre la cual se dibuja una proyección bidimensional de la geometría a visualizar.

La actualización del área de visualización de la escena en el visor tiene lugar de forma automática cada vez que parte de la geometría que se visualiza es modificada, o bien cuando tiene lugar un cambio en un nodo de selección asociado. La forma de actualización de la imagen puede ser configurada a través de la aplicación.

Open Inventor define distintos tipos de visores, que ofrecen distintos comportamientos de navegación al usuario. Así, podemos encontrar un visor de examen (*Examiner Viewer*) que permite girar alrededor de la escena. Un visor de paseo (*Walk Viewer*) hará que la cámara se desplace por la escena como si caminara sobre ella. Un visor de vuelo (*Fly Viewer*) ofrecerá una vista de pájaro de la escena, y permitirá al usuario moverse en un solo plano sobre ella.

La implementación de un visor depende del sistema operativo que dé soporte a la versión de Open Inventor que se esté utilizando. Si la aplicación está programada sobre entorno Unix, el visor estará relacionado con un *Widget* del sistema X-Windows. En caso de aplicaciones

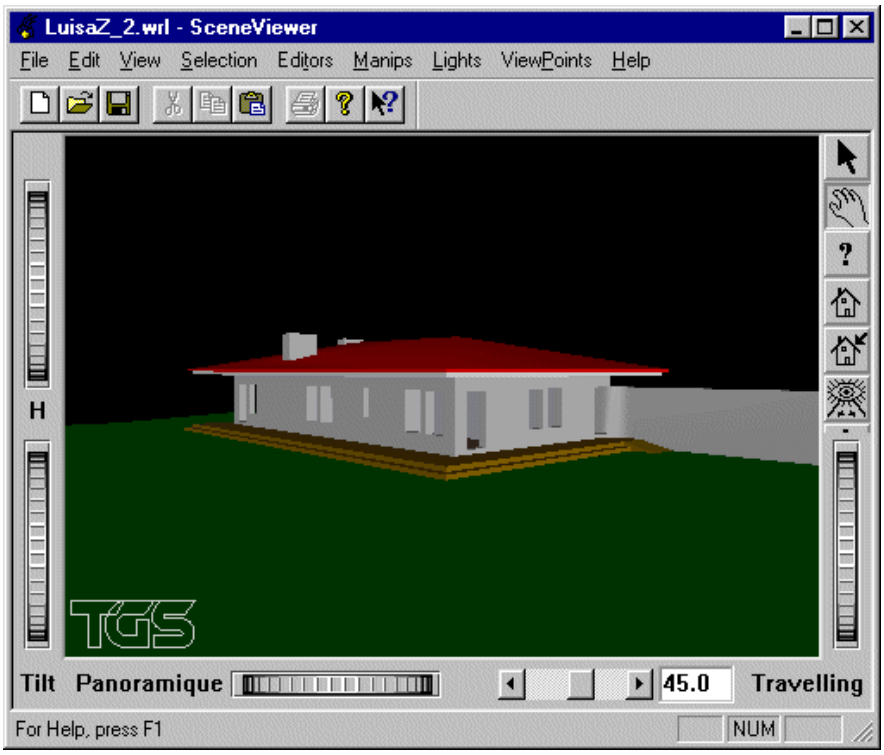

Figura 3.3.: Ejemplo de visor de Open Inventor.

programadas con la API de Microsoft Windows, el visor se relacionará con una clase de ventana. Finalmente, si se programa con las Microsoft Foundation Classes, el visor se relaciona con una vista de la aplicación.

En la Figura 3.3 se puede ver un visor de examen con los botones y controles asociados. La implementación de este visor es la de Windows NT, realizada por Template Graphics Software.

3.5. Relación entre Open Inventor y VRML 97

Merece la pena dedicar una sección a comentar las semejanzas y diferencias entre Open Inventor y VRML 97, ya que VRML es el formato con el que trabaja la aplicación. Por un lado, VRML 97 *no es* propiamente Open Inventor, con lo que no es posible trabajar de forma transparente con los dos formatos. Por otro lado, existen muchas relaciones y semejanzas entre los dos formatos, lo que hace posible que se puedan definir interacciones entre ellos.

En esta sección describiremos brevemente en qué consiste el formato VRML 97. Detallaremos las diferencias que el formato VRML 97 presenta con el formato propio de Open Inventor. Finalmente, explicaremos las funcionalidades que las últimas versiones de Open Inventor implementan para permitir y facilitar la gestión del formato VRML 97.

3.5.1. El formato VRML 97

El estándar ISO/IEC 14772, conocido como VRML 97 ([Car97], [Har96]), define un formato de fichero textual para la representación de escenas tridimensionales interactivas, junto con contenido multimedia. Su especificación se deriva de una propuesta de Silicon Graphics y presenta una filosofía básica similar al formato de fichero de Open Inventor, como veremos más adelante.

Entre los elementos más importantes que componen VRML 97, se pueden destacar los siguientes:

- Representación de mundos tridimensionales estáticos y dinámicos. Al igual que con Open Inventor, es posible representar escenas tridimensionales que interaccionen con el usuario.
- Posibilidad de incluir distintos tipos de elementos multimedia, tales como texto, imágenes, audio digitalizado, sonido sintético o secuencias de vídeo. Esto hace que el formato VRML 97 sea capaz de modelar entornos multimedia complejos.
- Hipervínculos entre distintos elementos de una misma escena y entre distintas escenas tridimensionales. Este aspecto permite el diseño estructurado de múltiples mundos VRML, entre los que el usuario puede navegar.
- Posibilidad de extender las funcionalidades de una escena mediante interfaces con lenguajes de *scripting* comúnmente utilizados, tales como JavaScript o VBScript. Esta extensión permite que se puedan programar funcionalidades complejas en una escena VRML, sin necesidad de depender del navegador para su implementación.

El formato VRML 97 modela la escena tridimensional como un grafo dirigido acíclico, igual que Open Inventor. Los nodos presentan comportamientos muy similares a los que han sido estudiados en la Sección 3.3.

3.5.2. Diferencias entre Open Inventor y VRML 97

En primer lugar, resaltemos una diferencia básica: bajo el título Open Inventor se engloban varios conceptos. Estos conceptos incluyen:

- Un conjunto de librerías gráficas de manejo de escenas tridimensionales.
- Una interfaz de programación de aplicaciones (API).
- Un formato de fichero que permite intercambiar escenas tridimensionales.

En cambio, VRML 97 *es sólo un formato de fichero*, lo que implica que la forma de manejarlo queda al arbitrio del programador. De esta manera, no es inverosímil que una aplicación Open Inventor pueda manejar ficheros VRML 97, ya que son simplemente un formato gráfico más.

Bajo esta perspectiva, y teniendo en cuenta que la comparación que tendrá lugar será entre *formatos de fichero* únicamente, pasamos a destacar las diferencias más notables entre ambos.

En el apartado anterior hemos constatado que existen semejanzas muy claras entre el formato definido por Open Inventor y el que define VRML 97. Las diferencias se encuentran sobre todo en las nuevas funcionalidades que introduce VRML 97, y sobre todo, en la organización y recorrido del grafo de escena. A continuación, explicaremos en qué consiste esta última diferencia fundamental.

Organización del grafo de escena y estado global

La diferencia más importante de VRML 97 con el formato de fichero de Open Inventor se encuentra en la *supresión de la propagación de atributos*. Esto significa que un nodo no puede modificar el estado global que afecte a otros nodos situados a la derecha y arriba en el grafo de escena. A nivel práctico esto se manifiesta en dos aspectos muy concretos: La especificación de atributos en el interior de los nodos de geometría, y el comportamiento de los nodos de agrupación.

A diferencia de la especificación de los atributos en Open Inventor, que tiene lugar mediante *nodos independientes* colocados a la izquierda o arriba, VRML 97 especifica los atributos en forma de *campos*. Cada nodo geométrico (implementado en Open Inventor como una clase del tipo *SoVRMLShape*) contiene un campo, implementado en forma de nodo, de tipo *SoVRMLAppearance*. Este nodo contiene los posibles atributos del nodo geométrico, atributos que pueden ser el *material* (*SoVRMLMaterial*), el *color* (*SoVRMLColor*), etc.

Open Inventor define nodos de agrupación que presentan comportamientos muy distintos respecto a la propagación del estado global de la escena (explicado en el Apartado 3.2.2). En cambio, los nodos de agrupación definidos en VRML 97 no permiten que el los atributos que forman parte del estado global afecten a nodos que se encuentren fuera de ellos. Es decir, su comportamiento estándar es como mínimo similar al que tiene el nodo *SoSeparator* de Open Inventor. No existe en VRML 97 ningún nodo con la funcionalidad equivalente al nodo *SoGroup* de Open Inventor, que únicamente recorre los hijos sin impedir que el estado global se modifique y propague.

Otra diferencia muy importante entre los nodos de agrupación de VRML 97 y los de Open Inventor radica en la forma en que se tratan los descendientes. De hecho, los descendientes de un nodo se encuentran *contenidos* en éste, como si de un campo más se tratara.

Para modelar las transformaciones que tienen lugar sobre una determinada geometría, VRML 97 define un nodo llamado *SoVRMLTransform*, que combina las funcionalidades del *SoSeparator* y del *SoTransform* de Open Inventor. Esto significa que las transformaciones geométricas definidas en este nodo afectarán sólo a los nodos geométricos que sean hijos de éste.

La decisión de no incluir la propagación de atributos en el formato VRML 97 responde a la posibilidad de mantener el formato de fichero más sencillo y fácil de optimizar. Por ejemplo, los nodos *SoVRMLTransform* permiten localizar de forma inmediata cuál es la transformación que afecta a un nodo determinado, ya que siempre es el nodo padre. Como contrapartida, debido a esta característica de diseño, los ficheros VRML 97 tienen tendencia a estar más profundamente anidados que los ficheros Open Inventor, lo que puede implicar más consumo de memoria en un recorrido recursivo.

3.5.3. Soporte para VRML 97 en Open Inventor

La empresa Template Graphics Software (TGS), que desde hace unos años es la que desarrolla nuevas versiones de Open Inventor, ha implementado soporte para el formato de fichero VRML 97 ([Hec97]). A continuación explicaremos brevemente en qué consiste este soporte.

Inclusión de nodos VRML

Open Inventor incluye desde su versión 2.4[5] soporte para nodos VRML 97. Cada uno de los nodos VRML 97 está implementado en forma de clase C++, como si de un nodo corriente se tratara. A nivel de nomenclatura, es fácil distinguir si un nodo es VRML o es Open Inventor, ya que los nombres de las clases que implementan los nodos VRML comienzan por el prefijo *SoVRML*. También ha sido implementado un sistema de herencia de nodos VRML, de forma que sea posible abstraer propiedades comunes entre los nodos Inventor y VRML, así como propiedades comunes a los nodos VRML.

La inclusión en Open Inventor de una gran mayoría de nodos VRML 97 facilita enormemente la tarea al programador, que no debe preocuparse de tratar estos nuevos nodos como un formato aparte. De hecho, podemos decir que el formato actual de Open Inventor comprende tanto los nodos estándar de Open Inventor como los nodos VRML. Se puede ver también al formato de Open Inventor como un "superconjunto" de VRML 97.

Funciones de conveniencia para nodos de agrupación

La noción de descendientes y ascendientes de un nodo cambia ligeramente en VRML 97 respecto a Open Inventor, ya que los hijos de un nodo están contenidos en campos de éste. Sin embargo, las librerías Open Inventor mantienen las mismas funciones de acceso y modificación de los hijos de un nodo, tales como *addChild*, *removeChild*, *getNumChildren*, etc. De esta manera, el manejo de los nodos hijos de un nodo de agrupación se hace más transparente al programador.

Acceso a la ruta de un nodo VRML 97

Una ruta permite acceder a la instancia concreta de un nodo, como se explica en la Sección 3.1. Esta estructura es muy importante, ya que permite diferenciar entre distintas instancias en el caso de nodos referenciados más de una vez, y además permite acceder a los ascendientes de un nodo.

Hemos comentado anteriormente, no obstante, que los nodos VRML no tienen "hijos" en el mismo sentido que los pueden tener los nodos de Open Inventor. Un hijo de un nodo de agrupación es un campo de éste. Por este motivo, la aplicación de una acción cualquiera que deba devolver un nodo VRML 97 devolverá una ruta que llegará únicamente hasta la profundidad del primer nodo VRML 97 encontrado, sin continuar con sus descendientes, ya que los considera campos de un nodo.

[5] La versión actual de Open Inventor es la 2.6.

Por este motivo, es necesario utilizar un nuevo mecanismo de Open Inventor, llamado *ruta completa* (*SoFullPath*). Una ruta completa se obtiene a partir de una ruta estándar realizando una conversión de tipos (*casting*). Esta nueva ruta contendrá la cadena completa hasta el nodo VRML 97 solicitado.

El mecanismo de ruta completa es utilizado también en otro contexto, los llamados *conjuntos de nodos* o *NodeKits* ([Wer94a]), que no son utilizados en este editor, y que por tanto no se explican en esta memoria.

Entrada y salida por fichero

Uno de los factores decisivos que hay que contemplar necesariamente en el soporte que una aplicación ofrezca al formato VRML 97, es la implementación de la entrada y salida por fichero. Es decir, la capacidad de leer y de exportar un fichero con formato VRML 97. En este apartado explicamos el soporte que Open Inventor ofrece para leer y escribir ficheros que contengan nodos VRML 97.

Gracias al soporte de las últimas versiones de Open Inventor, los nodos VRML 97 son aceptados en el formato Open Inventor como si de nodos estándar se tratara. Por tanto, la lectura de un fichero "mezclado" que contenga nodos VRML 97 y Open Inventor no presentará más complicación que la de uno que contenga únicamente nodos de Open Inventor. En caso de leer un fichero VRML 97 "puro", la última versión de Open Inventor proporciona una función que permite leer de fichero todo un árbol VRML y que retorna un nodo de agrupación VRML 97 (a diferencia de la función estándar, que retorna un nodo separador de Open Inventor). La capacidad de leer directamente un fichero que contenga nodos VRML 97 evita la necesidad de construir un analizador léxico-sintáctico que realice la conversión entre nodos VRML 97 y nodos de Open Inventor.

En el caso de la escritura, encontramos dos tipos de almacenamiento posibles:

- En formato Open Inventor, prácticamente equivalente a la representación interna del grafo de escena, con nodos mezclados Open Inventor y VRML 97.
- En formato VRML 97, de forma que se puedan exportar los datos a otras aplicaciones, con lo cual el fichero resultante no debe contener más que nodos VRML 97 válidos.

En caso de que el usuario desee guardar su trabajo en formato VRML 97, será necesario realizar una *conversión*, de forma que los nodos Open Inventor sean sustituidos por su equivalente VRML 97. Open Inventor implementa una *acción* (como las explicadas en el Apartado 3.2.1) llamada *SoToVRMLAction*. Esta acción permite convertir un árbol de escena que contenga *únicamente*[6] nodos Open Inventor a otro equivalente conteniendo sólo nodos VRML 97.

Carencias y errores en el soporte de Open Inventor para VRML 97

El hecho de que Open Inventor ofrezca soporte para un formato de fichero como VRML 97 no significa que este soporte sea perfecto. Existen diversas carencias, y diversos fallos que

[6] De hecho, hemos podido comprobar que la aplicación de esta acción a un subárbol que contenga nodos VRML 97 da lugar a un comportamiento indeterminado de la aplicación, y a una salida errónea.

complican el trabajo del programador, y que deben ser tenidos en cuenta en el momento de implementar nuevas funcionalidades que afecten a nodos VRML 97. Pasamos a enumerar a continuación algunas de las lagunas más importantes que hemos encontrado.

Existe una serie de nodos VRML 97 que no han sido implementados en Open Inventor. En el momento de recibir un fichero que contenga algunos de estos nodos, Open Inventor los leerá como "nodos desconocidos" y no será capaz de procesarlos automáticamente. Esto significa que el programador debe extender el soporte por su cuenta si quiere que éste incluya completamente el formato VRML 97.

Como hemos comentado más arriba, la aplicación de una acción que retorne nodos VRML 97 da como resultado una ruta incompleta. En caso que esta ruta sea visible desde la aplicación, es posible realizar una conversión de tipos. Sin embargo, Open Inventor aplica acciones de forma interna, como es el caso del redibujado (*rendering*). En el transcurso de la aplicación de una de estas acciones, éstas pueden retornar cadenas de índices incompletas, lo cual provoca que la aplicación funcione incorrectamente.

Aunque Open Inventor proporciona una acción que permite convertir un árbol de escena propio al formato VRML 97, si el árbol contiene algún nodo VRML 97, el funcionamiento de esta acción es indeterminado y erróneo. En la versión 2.5.0, la aplicación de esta acción daba como resultado un grafo VRML 97 que contenía multitud de nodos *SoSeparator* de Open Inventor vacíos. Esto provocaba que el fichero resultante no pudiera ser reconocido por navegadores de VRML. Con la nueva versión 2.5.2, al aplicar esta acción a un nodo VRML 97, el resultado es un nodo de agrupación de VRML 97 vacío.

4. Soporte al trabajo cooperativo

El editor ha sido concebido como una herramienta del tipo CSCW (*Computer-Supported Cooperative Work*, trabajo cooperativo asistido por ordenador [VV.94]), al igual que el Sistema M3D. Por tanto, presenta una serie de características comunes a todos los sistemas CSCW, que comentamos a continuación.

La característica principal de los sistemas CSCW es la interacción entre varios usuarios, de forma que sea posible realizar una actividad en grupo. Este trabajo en grupo es posible gracias al uso de diversas herramientas complementarias, que facilitan la comunicación entre los usuarios. Entre estas herramientas de soporte, podemos citar la comunicación textual (el comando *talk* de los sistemas Unix, por ejemplo), la audioconferencia y la videoconferencia.

Por otro lado, cualquier herramienta de producción que se pretenda integrar en un entorno CSCW debe presentar las siguientes características:

- Usuarios físicamente dispersos deben trabajar en un entorno compartido.
- La información contenida en ese entorno compartido debe ser protegida por mecanismos de control de acceso.
- La coordinación de actividades debe ser facilitada por el uso de canales de comunicación apropiados.

En este capítulo explicamos la estructura de soporte al trabajo cooperativo que ha sido implementada en el editor, así como las decisiones de diseño que se han tomado. En primer lugar, describiremos la plataforma básica de gestión CSCW, llamada JESP (*Joint Editing Service Platform* [Alm95]). A continuación, hablaremos de la estructura abstracta de replicación en memoria, llamada base de datos persistente. Finalmente, introduciremos el protocolo de aplicación Mu3D, utilizado para el envío de mensajes entre las réplicas de los datos.

4.1. La plataforma JESP

El diseño de un sistema colaborativo puede estar basado en dos enfoques principales. Uno de ellos consiste en desarrollar un sistema completamente dedicado al soporte de una o más aplicaciones concretas, de forma que se cree un entorno colaborativo con características específicas. El otro enfoque apunta a la creación de una plataforma que proporcione servicios de soporte genérico para aplicaciones distribuidas. Este último enfoque proporciona mayor flexibilidad, ya que los miembros de una sesión cooperativa pueden elegir qué herramientas utilizar, sin necesidad de modificar la plataforma de soporte. Por otra parte, la implementación

de las aplicaciones se puede mantener separada de la plataforma, de forma que el desarrollo de funciones adicionales sobre ésta se mantiene completamente transparente a las aplicaciones. De este modo, la arquitectura permanece abierta a desarrollos futuros.

La plataforma JESP, diseñada según este último enfoque, proporciona a las aplicaciones (y en concreto a nuestro editor) el soporte necesario para hacer posible las operaciones cooperativas. Esta plataforma ha sido diseñada para proporcionar dos tipos diferentes de servicios: control de sesión y comunicación de grupo. Estos servicios están disponibles para las aplicaciones a través de una API común. Los mecanismos de control de sesión liberan a la aplicación del coste de implementar las funciones necesarias para la inclusión en entornos cooperativos. Los servicios de comunicación ocultan a las aplicaciones la configuración punto-a-multipunto de bajo nivel.

En esta sección explicaremos la estructura de la plataforma JESP, así como los módulos dedicados a implementar el control de sesión y la comunicación, y la API implementada para interactuar con el editor.

4.1.1. Estructura general

Los mecanismos de control de sesión y de comunicación están organizados en una estructura por capas, como se muestra en la Figura 4.1. La implementación de estos grupos de operaciones se ha realizado mediante el desarrollo de dos módulos independientes, que se ejecutan como procesos separados en cada una de las máquinas que participan en la sesión cooperativa.

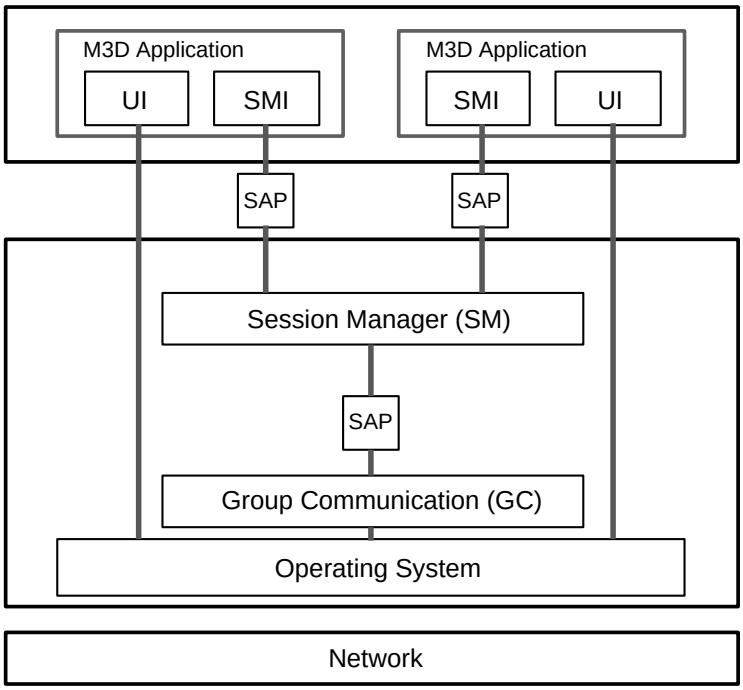

Figura 4.1.: Estructura modular de la plataforma JESP, y su relación con las aplicaciones.

Como se puede ver en la figura, los dos módulos encargados de las operaciones necesarias para el trabajo cooperativo reciben el nombre de *Session Manager* (SM), para las operaciones de control de la sesión, y *Group Communication* (GC), para las operaciones relativas a la comunicación punto-multipunto.

Cada aplicación distribuida puede utilizar un protocolo de aplicación específico para el intercambio de mensajes (en el caso del editor, el protocolo Mu3D). Estos mensajes son encapsulados por la aplicación y enviados al resto de sitios mediante una petición de servicio realizada al módulo SM. El módulo de la aplicación que estructura la comunicación entre la capa de aplicación y la capa SM recibe el nombre genérico de SMI (*Session Manager Interface*).

Los módulos de la plataforma interactúan con el sistema operativo mediante llamadas a los servicios que proporciona la pila de protocolos TCP-IP, presente en la práctica totalidad de sistemas operativos modernos.

4.1.2. Modo de ejecución

La plataforma JESP distingue entre dos tipos de usuarios, según el rol que ocupen en la sesión cooperativa. Estos dos tipos de usuarios reciben el nombre de *servidores* e *iniciador*. Explicamos a continuación en qué consiste esta diferenciación:

- El iniciador se encarga de enviar la orden de inicio a las demás réplicas de la plataforma.
- Los servidores se ejecutan y quedan a la espera de la orden de inicio por parte del iniciador. Una vez esta orden ha sido recibida por todos los servidores, la sesión cooperativa queda constituida.

De acuerdo con la estructura de capas de la plataforma JESP, el módulo GC es necesario para la ejecución del módulo SM. Por ello, su ejecución debe iniciarse primero.

Cada usuario en la plataforma JESP se identifica mediante un nombre de usuario (*username*) y una estación de trabajo (*host*). Éstos deben estar registrados previamente en un listado de miembros. Este listado de miembros está contenido en un fichero de texto llamado *JESPMembers*. Cada línea del fichero JESPMembers contiene, entre otros, los siguientes campos:

- El nombre de dominio de la estación de trabajo.
- El nombre del usuario correspondiente.
- Un número secuencial único, que identifica el par formado por el nombre del usuario y el nombre de la estación de trabajo.

Cuando un usuario inicia la plataforma JESP, se identifica con el nombre que tenga asignado. El programa comprueba entonces que el par formado por el nombre de usuario y la estación de trabajo con la que se conecta se encuentren en el fichero JESPMembers.

4.1.3. Interfaz Editor - JESP

La comunicación entre el editor y la plataforma JESP ha sido implementada mediante una interfaz de programación de aplicaciones (API). En este apartado describimos las funciones principales de esta API, y su funcionamiento.

Las funciones de interfaz entre la aplicación y la plataforma JESP se pueden agrupar en tres tipos de operaciones: inicialización de la plataforma, envío y recepción de mensajes, y gestión de los usuarios del grupo de trabajo cooperativo. Listamos a continuación las funciones utilizadas en cada tipo:

- Funciones de inicialización de la plataforma.
  - *JSPInitPlatform*. Realiza la configuración de la plataforma JESP, e inicializa las estructuras de datos necesarias.
- Funciones de interfaz de la aplicación con el módulo SM:
  - *JSPSendMsg*. Función de envío de mensajes estándar.
  - *JSPSendSimpleMsg*. Función de envío de mensajes cortos, sin parámetros.
  - *JSPQueryTeleEvent*. Función de lectura de los eventos remotos recibidos.
- Funciones de gestión de usuarios en el grupo de trabajo cooperativo:
  - *JSPGetGroupMemb*. Función de consulta de la lista interna de miembros de la sesión.
  - *JSPAddGroupMemb*. Función de actualización de la lista interna de miembros de la sesión.
  - *JSPFindMembID*. Función de consulta del identificador de usuario en la lista interna de miembros de la sesión.

El editor, al igual que la plataforma JESP, precisa que se indique el rol del usuario al inicio de la ejecución. Es decir, los usuarios deberán indicar si son servidores o el iniciador. Las instancias del editor que actúen como servidores quedarán a la espera de que el iniciador indique que pueden comenzar.

4.2. La base de datos persistente en memoria

Una de las decisiones más importantes en el diseño de un sistema CSCW es la elección entre una arquitectura centralizada y una arquitectura replicada. Explicamos a continuación en qué consiste cada una, sus ventajas e inconvenientes, y la decisión tomada en el desarrollo del editor.

Una *arquitectura centralizada* consiste en mantener una única copia de la aplicación ejecutándose en un servidor, de forma que todos los eventos producidos por los clientes son enviados

al servidor, y la salida de éste es distribuida a todos los clientes. El estado de la aplicación se puede mantener fácilmente consistente, ya que todas las modificaciones producidas por los usuarios son serializadas. Los cambios en el estado afectan solamente a una copia de los datos. El principal inconveniente de este esquema es el alto consumo de ancho de banda, y los retrasos en la respuesta, que resultan del doble trayecto del mensaje enviado por el cliente al servidor, y de la respuesta devuelta por el servidor al cliente. Un esquema de esta arquitectura se puede observar en la Figura 4.2.

En el caso de la *arquitectura replicada*, cada sitio en la sesión de trabajo ejecuta una copia de la aplicación. Todos los eventos producidos por los usuarios son procesados localmente y distribuidos a los otros sitios. Con este sistema, el tráfico sobre la red contiene solamente comandos, de modo que el tiempo de respuesta es mucho menor, y el ancho de banda requerido es mucho más pequeño que con el enfoque precedente. El principal problema de esta configuración es la necesidad de mantener la consistencia entre todas las copias de la aplicación. Un esquema de esta arquitectura puede observarse en la Figura 4.3.

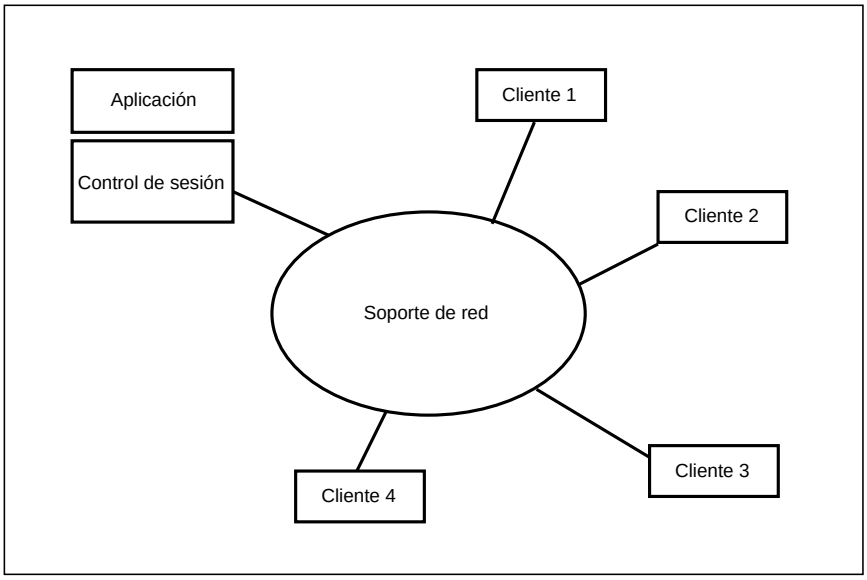

Figura 4.2.: Esquema de una arquitectura CSCW centralizada.

Uno de los requerimientos principales en el desarrollo del editor es la capacidad del sistema de ejecutarse a través de redes de ordenadores de ancho de banda medio o bajo. En concreto, se ha buscado un rendimiento aceptable en redes RDSI, que constituyen un estándar de conexión en pequeñas y medianas empresas en toda Europa.

Otro requerimiento básico, implícito en la propia naturaleza de los datos, es el acceso rápido a los datos. La visualización o *rendering* de datos tridimensionales comporta un volumen de datos muy elevado. Si esa visualización se debe realizar a frecuencias que aseguren una cierta interactividad (al menos 10 frames por segundo), el acceso rápido a todo ese volumen de datos es crucial.

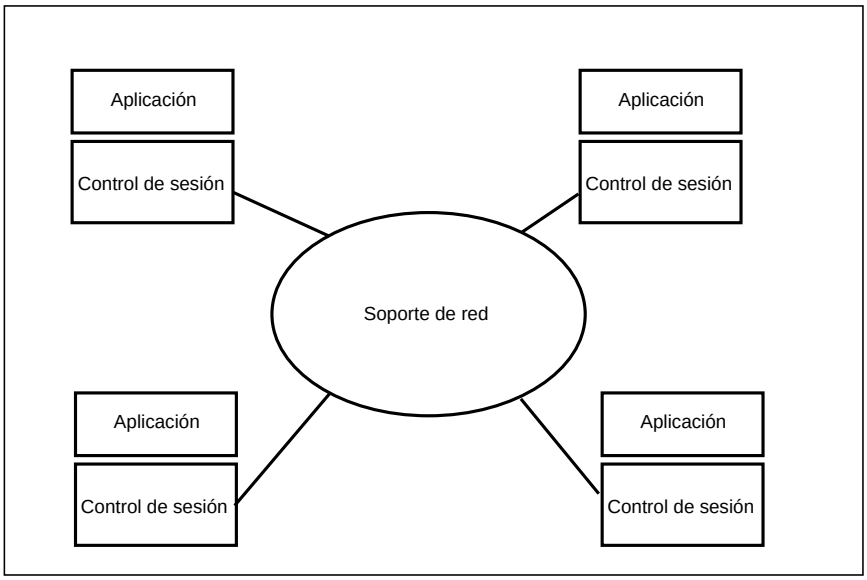

Figura 4.3.: Esquema de una arquitectura CSCW replicada.

Teniendo en cuenta estos requerimientos, se ha optado por una arquitectura de datos totalmente replicada en los sitios de ejecución. Los grafos de escena se hallan replicados en cada una de las instancias del editor y forman un esquema de memoria compartida distribuida, de forma que cada réplica constituye una copia coherente de una base de datos conceptual, que ha sido llamada "Base de datos persistente en memoria".

Cada una de las réplicas del editor contiene una copia de los datos geométricos tridimensionales. El acceso a esos datos, por tanto, es tan rápido como permita el *hardware* de visualización gráfica de la estación de trabajo, y no depende de la posible sobrecarga de la red.

La base de datos persistente en memoria es construida simétricamente en el momento que todas las réplicas de la aplicación son iniciadas. Al inicio de la ejecución, por tanto, cada réplica contiene exactamente los mismos datos relativos al diseño arquitectónico.

Dado que no existe una copia central de los datos geométricos, todas las réplicas se deben mantener coherentes a lo largo de la ejecución de la aplicación. Este es un aspecto básico, ya que cualquier funcionalidad nueva del editor deberá ser implementada respetando en todo momento la coherencia entre los datos replicados.

En la siguiente sección explicaremos la forma de envío de estos mensajes, su estructura, y el protocolo de aplicación que les da soporte.

4.3. El protocolo Mu3D

En la sección anterior hemos visto como las réplicas de la base de datos persistente en memoria deben permanecer coherentes a lo largo de la ejecución de la aplicación. Esta coherencia se consigue mediante el envío de mensajes entre las réplicas cada vez que una de ellas sufre una

modificación. Por ese motivo, el editor ha utilizado un protocolo de mensajes, llamado Mu3D ([SD97], [Gal00]).

El protocolo Mu3D es un protocolo de nivel de aplicación, que constituye un elemento esencial para el soporte de la interacción entre varios usuarios en el entorno compartido de la aplicación. Este protocolo es implementado mediante el intercambio de PDUs (*Protocol Data Units*) entre las réplicas de la aplicación.

Los cambios locales en el entorno compartido son notificados al resto de las réplicas mediante mensajes de actualización, que son enviados de forma asíncrona, en forma de eventos. El formato de estos mensajes es definido por el protocolo. De este modo, cada réplica de la aplicación es capaz de modificar los parámetros de la escena compartida. El trabajo de gestión de los eventos, desde el punto de vista del editor, comprende dos tareas:

- *Creación local del evento*. Cuando una modificación ha sido realizada sobre los datos tridimensionales, la aplicación encapsula los valores de ésta en un nuevo mensaje Mu3D, y los transmite al resto de réplicas.

- *Recreación de eventos remotos*. Cuando una réplica de la aplicación recibe un mensaje, lo decodifica e interpreta, y modifica los parámetros adecuados de su copia local de los datos tridimensionales.

Los mensajes que constituyen el protocolo Mu3D deben ser de un tamaño lo más pequeño posible, de forma que el ancho de banda consumido por el tráfico de mensajes sea mínimo. Este es un requerimiento crítico, dado que un mayor tamaño de los mensajes implica un mayor tiempo de transferencia. Este mayor tiempo de transferencia puede comprometer seriamente la interactividad de la aplicación. Si un usuario modifica un diseño arquitectónico y su modificación no es transmitida a los demás usuarios en un tiempo razonable, o peor aún, no es transmitida en absoluto, el trabajo cooperativo se vuelve imposible. Por este motivo, los mensajes del protocolo Mu3D contienen la información mínima necesaria para que el destinatario reproduzca localmente los eventos remotos, representados por los mensajes que llegan. En la implementación actual, el tamaño medio de un mensaje Mu3D oscila entre 130 y 150 bytes[1]. Este pequeño tamaño, junto con la técnica de replicación de los datos tridimensionales, proporciona un tiempo de respuesta más rápido, ya que la recreación local de un evento remoto no depende del tráfico de red.

Cada mensaje Mu3D consta de diversos campos divididos en cabecera y datos, como se puede ver en la Figura 4.4. Describimos a continuación cada uno de ellos:

- El campo USER ID contiene el identificador del usuario que envía el mensaje.

- El campo EVENT describe el tipo de evento. De acuerdo con este tipo de evento, el resto de campos pueden tener un conjunto de valores diferente, así como un significado distinto.

[1] En la implementación actual, parte de los mensajes son enviados íntegramente con representación en texto ASCII. Es posible reducir aún más el tamaño de los mensajes empaquetándolos, o enviando los datos numéricos en representación binaria.

- El campo LOCAL TS (*Local Timestamp*) se utiliza para la ordenación de los mensajes.
- El campo SCENE ID contiene el identificador de la escena sobre la que el evento ha sido generado.
- El campo SIZES contiene los tamaños de los campos de datos del mensaje.
- El campo OBJECT CLASS describe la clase de objeto a la cual se aplica el evento. Sus valores pueden variar de acuerdo con el tipo de evento, y suelen tener relación con el tipo de nodo a modificar.
- El campo OBJECT PATH contiene la ruta Open Inventor de índices que identifica al objeto afectado por el evento.
- El campo OBJECT DATA contiene datos variables, relacionados con el evento y con la clase de objeto del mensaje.

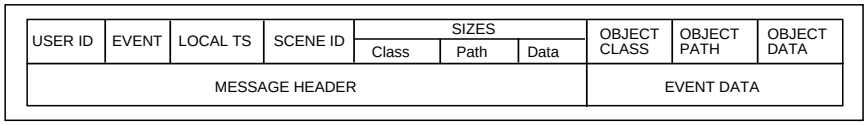

Figura 4.4.: Formato de un mensaje Mu3D.

El significado concreto de cada campo depende del tipo de evento que ha generado el mensaje, y de la clase de objeto a la que éste es aplicado, como se puede ver en la Tabla 4.2.

El conjunto de mensajes del protocolo Mu3D se divide en distintos tipos, según el área de aplicación de cada uno. Los enumeramos a continuación:

- *Mensajes de control de la sesión*. Utilizados para controlar el inicio de una sesión cooperativa.
  - TELEINIT. Indica que uno de los usuarios quiere iniciar una nueva sesión.
  - TELENOTIFY. Indica que un usuario se ha unido a la sesión.
- *Mensajes de control de los objetos*. Utilizados para implementar el bloqueo de objetos para su edición posterior.
  - SELECT. Un usuario ha seleccionado un objeto.
  - DESELECT. Un usuario ha liberado un objeto que tenía seleccionado.
  - SELECT NACK. Un usuario ha rechazado la selección de un objeto por parte de otro.
- *Mensajes de control de escenas*. Utilizados para la implementación de las operaciones básicas sobre escenas. Consisten en un único evento, SCENE, con las siguientes clases:

  - SCENE-NEW. Un usuario ha creado una nueva escena.
  - SCENE-REMOVE. Un usuario ha borrado una escena de memoria.
  - SCENE-ACTIVATE. Un usuario ha cambiado a otra escena.

- *Mensajes de edición*. Utilizados para la notificación de modificaciones sobre la escena tridimensional.

  - ADD. Un nuevo objeto ha sido añadido a la escena.
  - REMOVE. Un objeto ha sido borrado de la escena.
  - MODIFY. Una parte de la escena ha sido modificada.
  - CLIPBOARD. Una operación de portapapeles ha sido aplicada.
  - RPC. Llamada remota a un procedimiento complejo.

Una descripción detallada de los valores de cada tipo de mensaje del protocolo Mu3D se puede observar en la Tabla 4.2.

El protocolo Mu3D, que en un principio constaba de un conjunto mínimo de clases de mensajes, ha demostrado ser ampliable y flexible. Como veremos en los siguientes capítulos, nos ha sido posible extender el conjunto de tipos de mensajes, en el momento de introducir una nueva función o característica en el editor, sin modificar la estructura del protocolo.

Evento	Clase	Datos	Acción
ADD	FILE	Nombre fichero	Inserta una escena desde un fichero.
	AVATAR	Nombre fichero	Inserta un avatar en la escena.
	URL	URL	Inserta una escena desde la web.
	SECTION	Nombre	Añade una sección a la escena.
	ANNOTATION	Texto o enlace	Inserta una nueva anotación en forma de texto o enlace.
	RESTORE	ID de usuario	Restaura un objeto de la papelera.
	VIEWPOINT	Parámetros	Inserta un nuevo punto de vista.
	Otros	Tipo de nodo	Crea e inserta un nuevo nodo.
REMOVE	Otros		Borra el nodo de la escena.
	LIGHT	Nodo de luz	Borra la luz dada de la escena.
MODIFY	AVATAR	Transformación	Modifica la posición del avatar.
	NAME	Cadena	Cambia el nombre del nodo especificado.
	TRANSFORM	Transformación	Modifica la transformación del objeto.
	MATERIAL	Material	Modifica el material del objeto.
	ANNOTATION	Texto	Modifica el texto o URL de la anotación.
	SECTION	Ruta	Modifica los planos de corte de la sección.
	VIEWPOINT	Parámetros	Modifica un punto de vista de la escena.
	LIGHT	Nodo de luz	Modifica los parámetros de la luz.
CLIPBOARD	COPY	Objeto	Copia el objeto al portapapeles.
	PASTE	Objeto	Pega el objeto del portapapeles.
SCENE	NEW	ID de usuario	Crea una nueva escena vacía y le asigna un identificador.
	REMOVE	ID de escena	Borra la escena de la memoria.
	ACTIVATE	ID de escena	Coloca el avatar remoto en la escena.
	NAME	Cadena	Cambia el nombre de la escena.
SELECT	GROUP	Objeto	Marca el objeto como seleccionado.
SELECT NACK	USER	Booleano	Rechaza una selección por conflicto.
DESELECT	GROUP	Objeto	Libera la selección del objeto.
TELEINIT	USER	ID de usuario	Inicializa una sesión de trabajo e informa a todos los participantes.
TELENOTIFY	USER	ID de usuario	Respuesta de un nuevo participante.
RPC	GROUP	Objeto	Agrupa el subárbol según un criterio determinado.
	MOVE	Objeto	Mueve un objeto a otra localización en el grafo de escena.

Tabla 4.2.: Estructura y principales eventos del protocolo Mu3D

Parte II.

Técnicas cooperativas desarrolladas

5. Manejo cooperativo de los datos 3D

5.1. Definición de objeto

Un “objeto” es la unidad mínima utilizable por el usuario en nuestro programa. Un objeto es un subgrafo de escena que contiene los elementos geométricos mínimos para ser manipulado o editado independientemente del resto del grafo de escena. Por ello, la raíz de este subgrafo es un nodo separador. Cada objeto independiente contiene su propia geometría, su propia matriz de transformación y su propio nodo de material. Un ejemplo de objeto mínimo se puede observar en la Figura 5.1.

La granularidad de las operaciones del editor es como mínimo a nivel de objeto. Por tanto, no es posible seleccionar y modificar nodos aislados del grafo de escena. Todas las modificaciones que tengan lugar sobre objetos estarán encapsuladas mediante operaciones de usuario de alto nivel: modificaciones geométricas, edición de materiales, etc.

Los objetos pueden estar agrupados, como ya hemos visto, formando una jerarquía de árbol. Las características de un grupo de objetos pueden ser modificadas, de forma que la modificación afecte a todos los objetos del grupo, que se encuentran por debajo y a la derecha.

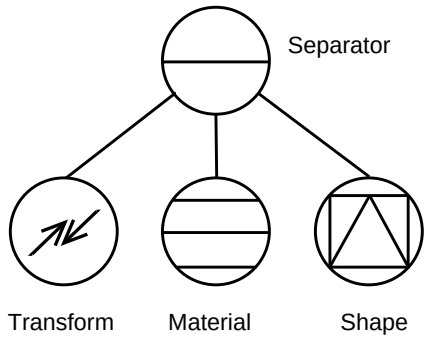

Figura 5.1.: Estructura mínima de un objeto tridimensional utilizable por el editor

5.2. Gestión cooperativa de escenas múltiples

La gestión de escenas múltiples es una característica muy extendida en los editores de objetos 3D. Esta característica permite incrementar la productividad, ya que permite al usuario trabajar

a la vez sobre diseños tridimensionales distintos.

Por escena entendemos un conjunto de datos geométricos interrelacionados, que pueden ser visualizados y modificados por el usuario. Cada escena puede estar compuesta de uno o más objetos, que se irán añadiendo a ella de forma acumulativa.

El editor permite gestionar múltiples escenas, asociando cada una de ellas a una o más ventanas. De este modo, el usuario puede trabajar con varias escenas en distintas ventanas, teniendo en todo momento una única escena activa.

Las escenas múltiples se encuentran integradas en el editor formando un único grafo de escena. Cada escena se asocia a una o más ventanas, de forma que la raíz de su subárbol asociado se asigna a los visores Open Inventor correspondientes.

La gestión de múltiples escenas exige la definición de un conjunto de operaciones. En nuestro caso, este conjunto de operaciones contendrá las siguientes:

- *Creación de una escena en memoria*. Esta operación supone la creación de una nueva escena, bien vacía, o bien conteniendo detos leídos de un fichero Open Inventor o VRML.

- *Eliminación de una escena*. Esta operación consiste en el borrado de memoria de todos los datos relacionados con la escena.

- *Activación de una escena*. En un momento dado, sólo una escena estará activa a la vez. La activación de una escena implicará la desactivación de la anterior escena activa.

En esta sección mostraremos la estructura global utilizada para agrupar el conjunto de escenas, así como la estructura interna de cada escena individual. Explicaremos además las operaciones básicas sobre las escenas que hemos desarrollado, y el conjunto de mensajes del protocolo Mu3D que han sido añadidos para cubrir el aspecto cooperativo de la gestión de escenas múltiples.

5.2.1. Estructura del conjunto de escenas

El árbol que agrupa la estructura global del conjunto de las escenas se muestra en la Figura 5.2. En la figura se ilustra la organización general de las escenas, y la estructura interna de una de ellas. Describiremos en primer lugar la estructura general del grafo en su totalidad, y a continuación describiremos la estructura de cada escena en particular.

El árbol global agrupa bajo un nodo raíz común (llamado *Scene Manager* en la figura) todos los subárboles correspondientes a cada escena. Bajo la raíz global se encuentra la siguiente estructura:

- Un subárbol que contiene datos internos, comunes a todas las escenas, como son los Portapapeles (*Clipboard*) y las Papeleras (*Thrash Bin*).

- Uno o más subárboles que corresponden a cada una de las escenas cargadas en memoria en el momento actual.

Todas las escenas tienen como raíz un nodo de selección del tipo *Local Scene Manager* (LSM en la figura), utilizado para la gestión de los objetos seleccionados interactivamente por el usuario. Sobre este tipo de nodo y su implementación hablaremos más extensamente en el Capítulo 6.

Cada nodo LSM tiene como hijo un nodo de agrupación de tipo separador, llamado *Root*, bajo el cual se encuentra toda la estructura de la escena. Los hijos que contiene el nodo Root, por este orden, son los siguientes:

- Un subárbol que contiene todas las luces de la escena (*Lights*).
- Un subárbol con todos los puntos de vista predefinidos de la escena (*Viewpoints*).
- Un subárbol con elementos comunes de visualización de la escena, tales como ejes de coordenadas, y otros (*Common*).
- Un subárbol que contiene todos los avatares de los usuarios remotos en la sesión (*Avatars*).
- Un subárbol con todas las secciones y cortes de visualización de la escena (*Sections*).
- Un subárbol que contiene todos los objetos de la escena, llamado *Objects*. Bajo este subárbol estarán colocados todos los objetos geométricos incluidos en la escena.

Cada escena está identificada por un número secuencial, que se utilizará para el soporte cooperativo a la gestión de escenas múltiples, como veremos en los siguientes apartados.

5.2.2. Operaciones básicas sobre escenas

La aplicación, hemos dicho, define tres operaciones básicas a realizar sobre las escenas: creación, borrado y activación. A continuación, describimos la implementación de estas operaciones.

La *creación* de una nueva escena consiste en la inserción bajo la raíz del árbol general de la escena de un nuevo nodo de selección, de tipo LSM, bajo el cual colgará una estructura de árbol como la descrita anteriormente. El usuario que haya creado la escena la activará en ese momento. La nueva escena estará vacía al principio, es decir, bajo el subárbol de objetos no habrá ningún nodo. El usuario puede elegir entre crear la nueva escena vacía (operación File-New), o crearla conteniendo datos leídos de un fichero (operación File-Open), con lo que se insertarán uno o más objetos en la escena después de su creación.

El *borrado* de una escena consiste en el borrado del nodo raíz de selección de ésta. Todos los datos relativos a la escena son descargados de memoria, y se activa otra de las escenas que haya en memoria en ese momento.

La *activación* de una escena consiste en asignar el nodo de selección de la escena al visor activo de la aplicación, de manera que se visualice la nueva escena activa. Además, se debe modificar el indicador de la escena activa, para que apunte al identificador de la nueva escena.

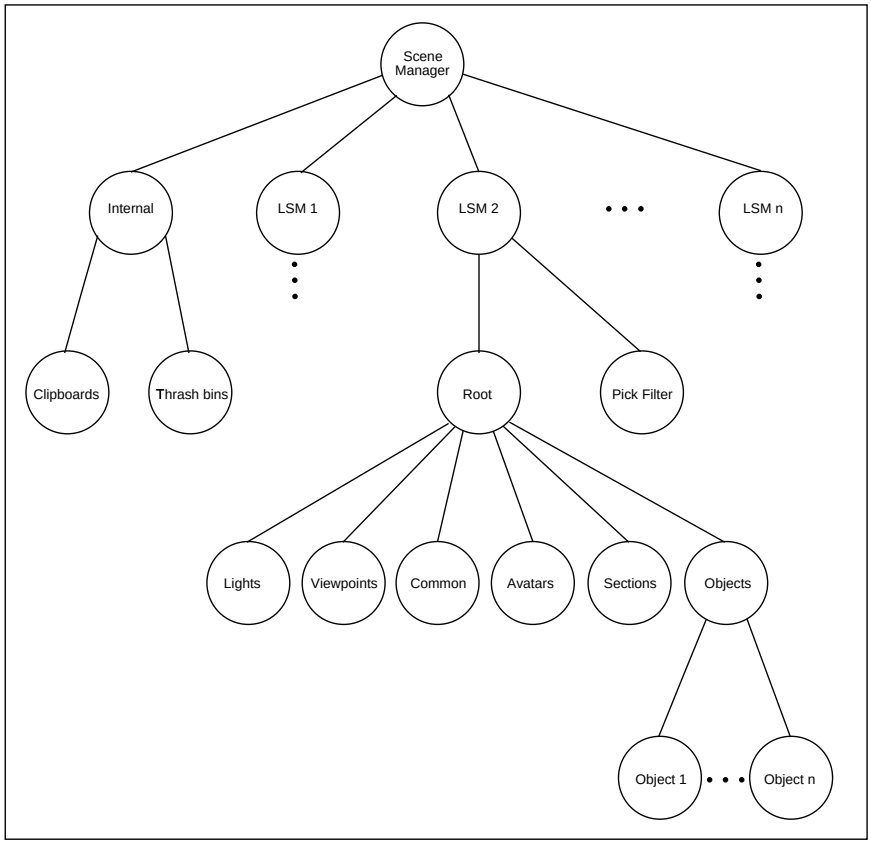

Figura 5.2.: Estructura del árbol de escenas del editor.

5.2.3. Soporte cooperativo a la gestión de escenas múltiples

Las operaciones sobre escenas descritas anteriormente deben comunicarse a todas las réplicas de la aplicación en la sesión. Por ello, el protocolo Mu3D ha sido extendido con un nuevo tipo de evento, llamado SCENE, que incluye distintas clases de eventos, una para cada operación. Veremos a continuación cuáles son estas clases, y el comportamiento de la aplicación al recibir un mensaje de estos tipos.

- El mensaje SCENE-NEW es enviado en el momento de la creación de una escena. Al recibir este mensaje, las otras instancias del editor recrean el evento, y activan la nueva escena.
- El mensaje SCENE-DELETE es enviado en el momento del borrado de una escena.
- El mensaje SCENE-ACTIVATE se envía en el momento que un usuario cambia de una escena a otra. El identificador de la escena de destino es enviado dentro del mensaje. Al recibir este mensaje, cada instancia del editor mueve el avatar del usuario origen a la nueva escena activa.

El borrado de una escena es una operación delicada, desde el punto de vista cooperativo. Por este motivo, hemos tenido que definir una serie de restricciones. Las describimos a continuación:

- Una escena no puede ser borrada si otros usuarios, además del que la va a borrar, se encuentran en ella, es decir, si la tienen activada. Para comprobar esto, se puede consultar el subárbol de avatares remotos de la escena, y ver si contiene algún elemento.
- Tampoco ha de ser posible borrar una escena si otros usuarios, aunque no se encuentren en ella, mantienen algún objeto seleccionado dentro de la misma. Esto se puede comprobar consultando la lista de objetos remotamente seleccionados del nodo LSM correspondiente, sobre la que hablaremos con detalle en el Capítulo 6.
- Por otra parte, una escena no debe poder ser borrada si no hay ninguna otra escena en memoria. Por tanto, la aplicación siempre tendrá cargada al menos una escena.

Como hemos explicado en la Sección 4.3, cada mensaje del protocolo Mu3D lleva incluido el identificador de la escena en la cual se ha generado el evento. Este campo fue añadido para que los datos contenidos en cada mensaje, tales como la ruta al objeto que se aplica, fueran relativos a la escena. Los eventos recibidos son recreados tomando como punto de partida la escena que contienen. De este modo, todas las escenas son actualizadas automáticamente, independientemente de cuál sea la escena activa de cada usuario en un momento dado.

5.3. Formatos tridimensionales soportados

El editor debe ofrecer soporte para los formatos más estandarizados de representación tridimensional. Este soporte incluye tanto la importación como la exportación de ficheros. El soporte que hemos implementado incluye los siguientes formatos:

- El formato *Open Inventor*, nativo de las librerías de desarrollo.
- El formato VRML 1.0, primera versión del estándar VRML.
- El formato ISO estándar VRML 97 ([Car97]), la versión más reciente del estándar.

Estos tres formatos son muy similares en la filosofía, ya que los tres se basan en el concepto de grafo de escena. Pasamos a continuación a explicar la relación entre ellos, desde el punto de vista de la conversión entre formatos.

- En primer lugar, podemos considerar, a efectos de la aplicación que hemos desarrollado, que los formatos Open Inventor y VRML 1.0 son prácticamente idénticos, ya que utilizan los mismos tipos de nodos. La única diferencia se encuentra en la cabecera del fichero, por lo que deberemos modificarla según el formato en el que se grabe en fichero.
- Por otra parte, como se explica en la Sección 3.5, el formato VRML 97 se diferencia de los dos anteriores principalmente en la estructura del grafo de escena, y en la utilización de nodos distintos, tanto en nomenclatura como en comportamiento.

Si el usuario trabaja habitualmente con otro formato, como por ejemplo DXF ([Rul96]), formato propio del programa AutoCAD [Aut90], que constituye un estándar *de facto*, o 3DS ([Rul96]), formato tridimensional del programa 3DStudio ([Aut94]), deberá realizar una conversión al formato VRML 97 o a Open Inventor. Esta conversión puede ser realizada con alguno de los múltiples programas comerciales que existen ([Cad]), o bien utilizando un módulo de conversión desarrollado dentro del proyecto M3D ([HF99]), que permite cargar estos tipos de ficheros en el editor, de forma transparente.

5.4. Entrada y salida

Las operaciones de lectura y almacenamiento de todo el grafo de escena o de una parte son independientes de la localización física de los datos. Por tanto, los mismos datos se pueden leer o guardar en un disco local a cada usuario, o bien en una base de datos centralizada, utilizando las mismas técnicas.

La entrada y salida son consideradas en el editor desde dos puntos de vista distintos: por un lado, hemos de determinar de qué forma se leen y escriben los datos tridimensionales. Por el otro, hemos de definir qué comportamiento tienen las operaciones de entrada y salida en el entorno cooperativo en que funciona el editor. A continuación explicamos de qué manera hemos implementado estos dos aspectos, tanto en la lectura como en el almacenamiento de datos.

5.4.1. Lectura

El principal problema relativo a los datos a leer viene determinado por el formato de éstos. El formato que presenta más diferencias de los tres formatos nativos es VRML 97, como hemos visto en la sección anterior. A continuación explicaremos de qué forma hemos implementado el soporte para la lectura de ficheros VRML 97.

Las últimas versiones de Open Inventor han incorporado soporte interno para nodos del formato VRML 97 ([Hec97]), por lo que manejar este formato no implica la construcción de analizadores sintácticos, ni de otras herramientas de conversión a la hora de cargar un fichero VRML 97. Sin embargo, ya hemos comentado que el manejo de nodos VRML 97 presenta numerosos problemas, debido a fallos en la implementación de Open Inventor, que no han sido solucionados por completo todavía.

Por estos motivos, hemos tomado la decisión de utilizar como formato interno únicamente el de Open Inventor (o VRML 1.0), de forma que aseguraremos que el grafo de escena sólo contenga nodos de estos formatos, cuando esté cargado en memoria.

No obstante, nuestro módulo de lectura aprovecha las funciones de análisis léxico y sintáctico de Open Inventor, que permiten cargar un fichero VRML 97 a un grafo de escena. Una vez el grafo de escena está cargado, es convertido a otro grafo de escena que contiene sólo nodos VRML 1.0 utilizando una función desarrollada para la ocasión, que recorre recursivamente el grafo de nodos VRML 97 y va generando un grafo con nodos VRML 1.0. Un esquema de esta operación se puede ver en la Figura 5.3.

La función de conversión de VRML 97 a VRML 1.0 aprovecha dos características básicas:

- La simplicidad estructural de los ficheros VRML 97, que permiten convertir fácilmente una estructura VRML 97 en una estructura VRML 1.0.
- La correspondencia que existe entre los nodos de los dos formatos, que permite encontrar un nodo equivalente VRML 1.0 para cada nodo VRML 97.

Esta función de conversión, presenta algunas carencias, entre las que podemos enumerar las siguientes:

- Falta de soporte al prototipado de nodos, una de las características más potentes de VRML 97, que no es soportado completamente por las librerías Open Inventor.
- Falta de equivalente VRML 1.0 para algunos nodos VRML 97.

No obstante, la función implementada ha demostrado ser suficiente para la totalidad de proyectos arquitectónicos utilizados como modelos de prueba.

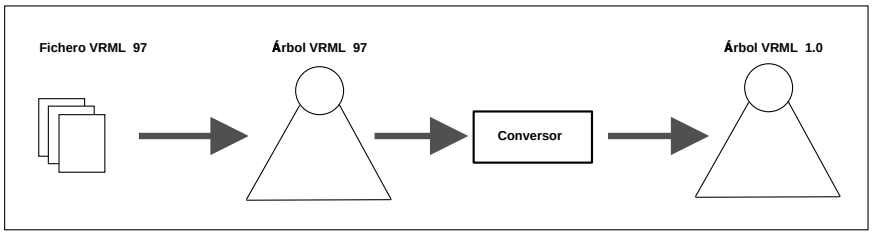

Figura 5.3.: Conversión de ficheros VRML 97 a Open Inventor.

La lectura de datos desde un fichero o base de datos debe ser implementada de forma cooperativa, de modo que los distintos usuarios realicen simultáneamente las lecturas. Para seguir esta filosofía, cada vez que una réplica del programa carga datos en memoria, envía un mensaje del protocolo Mu3D al resto de réplicas, de forma que hagan lo propio. Este mensaje, de tipo ADD, puede ser de la clase FILE, si los datos son cargados desde un fichero, o de la clase URL, si los datos son cargados desde una base de datos.

5.4.2. Almacenamiento

En un momento dado, un usuario puede guardar en el disco local o en una base de datos la escena activa, o bien una parte de ésta. La escena activa contiene una serie de información que no interesa guardar. Esta información incluye avatares, ejes, manipuladores, y otros elementos del grafo de escena, que no están replicados del mismo modo para todos los usuarios. Para mantener la coherencia entre las distintas réplicas, y para asegurar que la escena es guardada de igual modo por cualquier usuario, será necesario realizar un proceso de *filtrado* sobre ésta. Este filtrado consta de dos fases:

- En la primera fase se retiran todos los manipuladores del árbol de la escena, tanto los manipuladores de las luces como los manipuladores geométricos interactivos.

- En la segunda fase, se crea una nueva estructura de árbol temporal para el guardado. Para ello, se crea un nuevo nodo padre temporal, al que se añaden como hijos los distintos objetos que contiene la escena, así como el subárbol de las luces de la escena. Ese nodo temporal, y con él todos los hijos que se le han añadido, se guarda en un fichero, mediante una acción de escritura en disco de Open Inventor. Una vez ha tenido lugar la operación de guardado, el nodo temporal se borra, de forma que el árbol queda igual que antes del inicio de esta segunda fase. Un esquema de esta técnica se puede ver en la Figura 5.4.

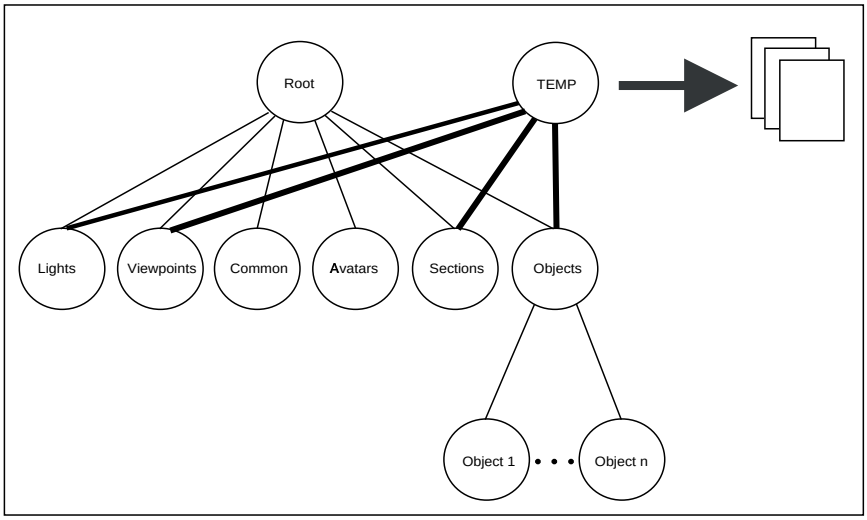

Figura 5.4.: Técnica del separador temporal utilizada para guardar el fichero.

Utilizando esta técnica, se guardan únicamente los objetos y las luces de la escena, sin los manipuladores ni la información local complementaria, que puede variar de usuario a usuario. De este modo, una copia almacenada en un momento dado, en disco local o en una base de datos, contendrá la misma información, independientemente del usuario que la haya guardado.

Para almacenar los elementos de una escena en formato Open Inventor o VRML 1.0 no es necesario realizar más operación que la inserción de la cabecera de fichero apropiada. Los nodos de ambos formatos son idénticos.

El almacenamiento de los elementos de una escena en formato VRML 97 hace necesaria la aplicación de una acción de conversión, llamada *SoToVRML2Action*, como se ha explicado en la Sección 3.5. Esta acción es aplicada a cada uno de los subárboles que contengan objetos geométricos, antes de ser guardados.

6. Acceso concurrente a los objetos 3D

El editor debe ser capaz de gestionar el acceso concurrente a los objetos 3D de una escena. La gestión de este acceso concurrente ha sido implementada mediante un mecanismo de selección. La selección es un aspecto crucial en el entorno cooperativo sobre el que funciona el editor. Este mecanismo garantiza la exclusión mutua entre los usuarios al realizar operaciones sobre los objetos tridimensionales que forman el grafo de escena, mediante una operación de *locking* (bloqueo) implícito. De esta manera, se evita que dos usuarios se sobreescriban entre ellos las modificaciones realizadas a un objeto.

En este capítulo explicaremos el concepto de selección, la forma como ha sido implementado en el editor, y los aspectos relacionados con la representación de los objetos seleccionados local y remotamente.

6.1. Concepto de selección

Una selección es la forma de especificar el objeto tridimensional sobre el que se van a realizar las operaciones. En el Capítulo 3 hemos explicado como Open Inventor representa la selección mediante un nodo de agrupamiento especial, llamado *nodo de selección*, que mantiene una lista con todos los objetos seleccionados por el usuario.

La implementación de la selección realizada por Open Inventor se limita a los objetos seleccionados por el usuario local. En el entorno cooperativo en el que ha de funcionar el editor, esta implementación es necesaria pero no suficiente. Por tanto, debemos ampliar este concepto al caso cooperativo y añadir una serie de condiciones adicionales a la selección:

- Un usuario sólo puede realizar operaciones de edición sobre un objeto si lo ha seleccionado previamente.
- Un mismo objeto sólo puede estar seleccionado simultáneamente por un usuario.

Estas restricciones vienen dadas por la necesidad de gestionar el acceso concurrente a cada objeto por parte de múltiples usuarios. Esto lleva a la necesidad de exclusión mutua entre usuarios cuando se está modificando un mismo objeto. En términos de concurrencia clásicos, podemos representar las modificaciones sobre cada uno de los objetos de la escena como una *región crítica* ([Sil94]), a la cual únicamente un usuario puede tener acceso al mismo tiempo.

En base a estas restricciones, hemos definido la siguiente *política de selección*:

> Cada vez que un usuario selecciona un objeto, se convierte en el *propietario* de este objeto, hasta que lo deselecciona. Durante este intervalo de tiempo, sólo ese

> usuario (el propietario) puede realizar operaciones de edición sobre el objeto, y ningún otro usuario puede realizar ediciones sobre éste, ni tampoco seleccionarlo.

Es decir, en términos prácticos, una selección de un objeto implica un bloqueo (*locking*) de éste. Esta política de bloqueo de objetos basada en selecciones nos permite mantener la consistencia entre todas las réplicas de la base de datos persistente. A continuación explicamos la forma en que esta política ha sido implementada en el editor.

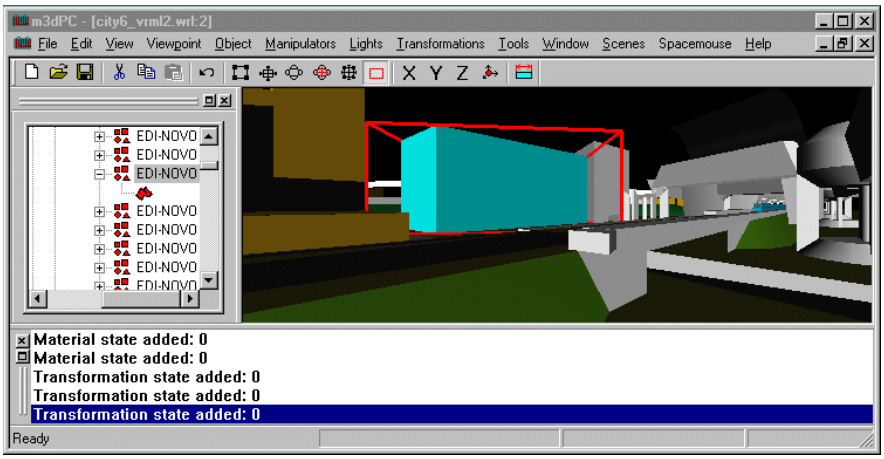

Figura 6.1.: Ejemplo de selección local resaltada en color rojo.

6.2. Implementación de la política de selección

El nodo de selección de Open Inventor, definido en el Capítulo 3, no es suficiente para satisfacer las restricciones que hemos definido en la sección anterior. Por ello, hemos implementado un nuevo tipo de nodo Open Inventor, llamado *LocalSM* (*Local Selection Manager,* gestor local de selecciones), como una nueva clase en lenguaje C++ que hereda las propiedades de la clase del nodo de selección, y que incorpora nuevas funcionalidades. Estas son las siguientes:

- Gestión de una lista de objetos seleccionados remotamente.
- Control de la posibilidad o no de seleccionar un objeto localmente.
- Envío de mensajes apropiados cuando un objeto es seleccionado o liberado localmente.

En la Sección 5.2 hemos explicado como cada una de las escenas cargadas por el editor en memoria tiene como raíz un nodo del tipo *LocalSM,* de manera que todas las selecciones y deselecciones que tienen lugar sobre una misma escena son procesadas por el mismo gestor local.

El gestor local de selecciones tiene dos cometidos principales: el control de las selecciones locales, y el control de las selecciones remotas. A continuación comentamos cómo hemos implementado las funciones necesarias para cubrir estos dos cometidos.

6.2.1. Control de selecciones locales

Open Inventor define la posibilidad de capturar los eventos de selección y deselección mediante funciones *callback*[1]. La aplicación aprovecha este soporte para gestionar las selecciones locales. En nuestro caso, hemos utilizado tres funciones diferentes, que son llamadas en caso de producirse los siguientes eventos:

- En el momento de pulsar o de soltar el botón del ratón sobre un determinado objeto (operación de *picking*).
- En el momento de seleccionar un objeto de la escena (operación de *selección*).
- En el momento de deseleccionar un objeto de la escena (operación de *deselección*).

Las funciones *callback* asociadas a los distintos eventos se han implementado como miembros estáticos[2] de la clase C++ *LocalSM*.

A continuación se detalla, de forma general, el comportamiento de la aplicación en el momento de invocación de cada una de las funciones *callback*.

Función de manejo de *picking*

La función miembro *SelectionPickCB* es invocada en el momento en que el usuario pulsa o libera el botón del ratón. En el caso más frecuente, en el que se hace clic sobre el mismo objeto sin apenas desplazar el ratón, la función será invocada dos veces con los mismos argumentos.

Esta función recibe como argumento una acción del tipo *SoRayPickAction*, que ha sido aplicada al objeto enviando un rayo en la dirección de vista de la cámara, y localizando el objeto u objetos que intersectan con este rayo. En el momento de la llamada a la función, los objetos se encuentran en la lista de resultados de la acción.

Tras realizar las operaciones adecuadas, la función retorna una *ruta* de Open Inventor que será pasada como parámetro al nodo de selección para ser añadida a la lista de objetos seleccionados localmente. El valor de la ruta a retornar puede ser modificado dentro de esta función, de manera que la selección se produzca sobre el objeto señalado por el usuario, sobre otro, o no se produzca.

En el momento en que la función de *picking* es invocada, se realiza una operación de *filtrado* de los objetos contenidos en la acción *SoRayPickAction*, que comprueba el tipo de nodo que son, y actúa de acuerdo con un conjunto de *reglas* establecidas. Este conjunto de reglas es extensible a medida que se van añadiendo nuevas funcionalidades a la aplicación, y en este momento son las siguientes:

- Si el objeto está seleccionado remotamente por otro usuario, no podrá ser seleccionado de nuevo localmente. Por tanto, la función retornará una ruta vacía, de forma que la selección no se realice.

[1] Una función *callback* no es llamada explícitamente por la aplicación, sino que es invocada por la librería de soporte o por el sistema operativo en el momento que sucede un determinado evento.

[2] El lenguaje C++ exige que las funciones *callback* no estén ligadas a ninguna instancia de las clases, y por eso estas funciones se han declarado como estáticas (*static*).

- Si el objeto es el manipulador de una luz, no se realizará ninguna operación de selección sobre ella. Por tanto, la función retornará una ruta vacía.
- Si el objeto ha sido definido como invisible[3], tampoco podrá ser seleccionado, y la función retornará una ruta vacía.
- Si no se cumple ninguna de estas reglas, el objeto es seleccionado, y la función retornará la ruta que lleva hasta el nodo de agrupación (*SoGroup*) más próximo a la geometría considerada.

Como podemos ver en esta última regla, la función de manejo de *picking* no devuelve el nodo sobre el que se ha posicionado el ratón, sino su ascendiente más cercano, de acuerdo con lo que se explica en la Sección 5.1.

El hecho de devolver la ruta del ascendiente más directo del nodo que el usuario ha señalado permite tener un referente fijo para poder operar con la selección. De este modo, aunque se modifique alguna de las propiedades del objeto (por ejemplo, insertando un nuevo nodo de material), la ruta que identifica al nodo seleccionado no cambiará, ya que la inserción tiene lugar por debajo de él.

Función de manejo de selecciones

La función miembro *SelectionCB* es invocada cada vez que un objeto es seleccionado. En el momento de la invocación, el objeto ya ha sido añadido a la lista de selecciones que mantiene el nodo de selección y es pasado a la función como parámetro. La operación que realiza esta función consiste básicamente en la notificación de la realización de esta selección a todos los demás miembros de la sesión cooperativa.

Dicha notificación tiene lugar mediante el envío de un mensaje Mu3D de tipo SELECT a todos los demás miembros de la sesión, incluyendo la ruta al objeto a seleccionar.

Función de manejo de deselecciones

La función miembro *DeselectionCB* se invoca cada vez que un objeto es deseleccionado. En el momento de la invocación, el objeto ya ha sido retirado de la lista de selecciones locales mantenida por el nodo de selección.

La función realiza el envío de un mensaje Mu3D de tipo DESELECT a todos los demás miembros de la sesión, incluyendo la ruta al objeto a deseleccionar.

6.2.2. Control de selecciones remotas

Una selección remota tiene lugar en una instancia del programa en la sesión cooperativa distinta de la instancia local. Cada vez que una selección tiene lugar en uno de los ordenadores miembro, la instancia del programa envía un mensaje notificándolo a los demás miembros, de manera que también ellos tengan constancia de esta selección.

[3] La capacidad de ocultar objetos en el editor ha sido implementada por separado, y no se explica en esta memoria.

Para almacenar las distintas selecciones remotas, cada gestor local de selecciones mantiene una lista con punteros a los objetos tridimensionales seleccionados remotamente. En el momento que llega un mensaje Mu3D de tipo SELECT o DESELECT, esta lista es actualizada. La consulta de la lista tiene lugar cada vez que un usuario intenta seleccionar localmente un objeto, para comprobar que ese objeto no está ya seleccionado en otra de las máquinas.

6.3. Representación gráfica de las selecciones

Un objeto seleccionado es representado en pantalla por Open Inventor mediante un resaltado (*highlight*). Este resaltado se visualiza mediante el contorno de la *bounding box* del objeto.

Para distinguir entre las selecciones realizadas localmente y las selecciones remotas, el editor utiliza dos colores: rojo para las selecciones locales, y amarillo para las selecciones remotas. Un ejemplo se puede ver en la Figura 6.2.

La representación en rojo de los objetos seleccionados localmente está implementada por defecto en Open Inventor. No ocurre lo mismo con la representación en amarillo, que ha tenido que ser implementada en nuestro editor. Explicamos a continuación el proceso que hemos seguido.

El resaltado del objeto es representado internamente como un objeto más de Open Inventor, formado por un separador, un nodo que representa la forma de la *bounding box*, otro que representa el color, y otro que indica que sólo se debe visualizar el contorno de las aristas (representación *wireframe*). Este objeto es creado por la acción de visualización cuando recorre el grafo de escena y encuentra un objeto seleccionado. La aplicación extiende este comportamiento de forma que se modifique el color según el tipo de selección, local o remota. Para ello hemos creado una nueva acción de visualización que realiza este comportamiento. La nueva acción ha sido implementada como una nueva clase C++ derivada de la clase *SoRenderAction*, siguiendo las técnicas descritas en [Wer94b]. Hemos llamado a esta clase *ownHighlightRenderAction*.

El funcionamiento de la nueva acción de visualización es muy similar en lo fundamental a la clase base de la cual hereda el comportamiento. La única diferencia radica en que mantiene un pequeño grafo Open Inventor, que contiene la estructura básica de una *bounding box* (geometría de la caja, color e indicaciones de visualización). Cada vez que la nueva acción de visualización atraviesa un nodo del grafo de escena, realiza las siguientes operaciones:

- Comprobar si el nodo se encuentra en la lista de selecciones locales.
  - Si es así, calcular su *bounding box*, aplicar sus medidas a la caja de resaltado, y dibujarla en color rojo, en modo *wireframe*.
- Comprobar si el nodo se encuentra en la lista de selecciones remotas.
  - Si es así, calcular su *bounding box*, aplicar sus medidas a la caja de resaltado, y dibujarla en color amarillo, en modo *wireframe*.
- Aplicar la acción base de visualización al nodo.

La nueva acción de visualización y resaltado es asignada a cualquiera de los visores del editor mediante el método *setGLRenderAction*. De este modo, la acción es aplicada automáticamente cada vez que la geometría es modificada, o bien cuando una selección local o remota es activada o desactivada.

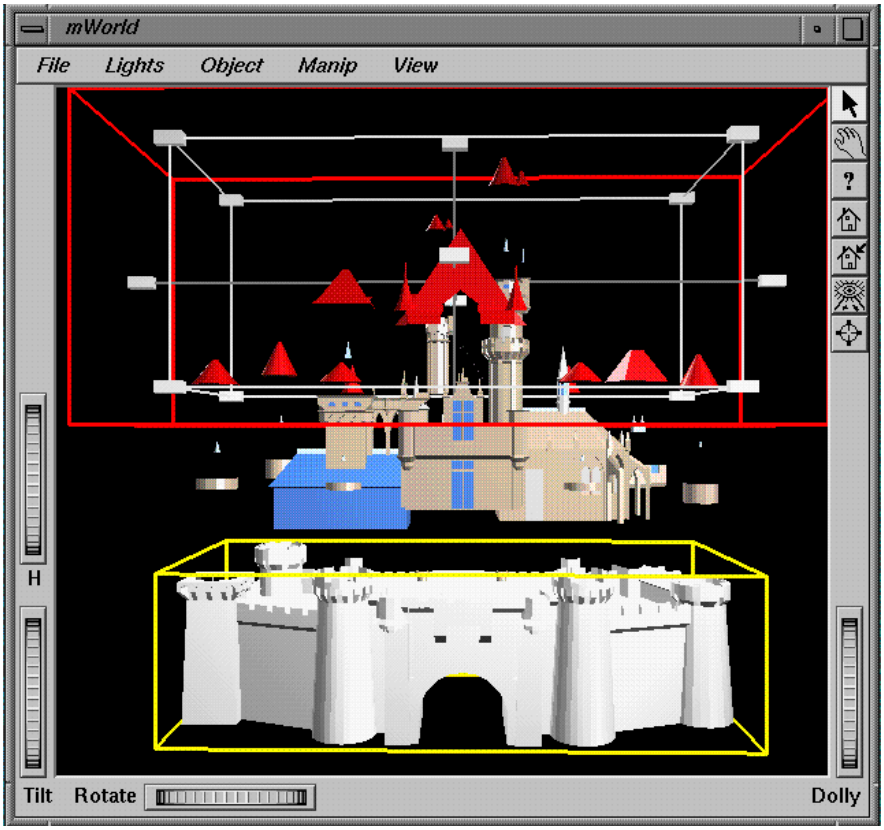

Figura 6.2.: Selección local (en rojo) y selección remota (en amarillo).

7. Edición básica cooperativa

Las funciones básicas de edición presentes en cualquier editor de objetos tridimensionales han sido implementadas también en la aplicación. El desarrollo de estas funciones básicas se ha realizado teniendo presente el entorno cooperativo en el que se ejecuta la aplicación.

En este capítulo presentamos la implementación de las funciones básicas del editor: inserción y borrado de objetos, manipulación geométrica interactiva, edición de materiales, e iluminación. Hemos adaptado cada una de estas funciones al trabajo cooperativo, utilizando el protocolo Mu3D y el concepto de memoria compartida distribuida con replicación activa, ambos explicados en el Capítulo 4. A continuación explicamos la forma en que lo hemos hecho.

7.1. Requerimientos generales

El desarrollo de cualquier función de edición sobre los objetos tridimensionales debe cumplir una serie de condiciones desde el momento en que esta función debe realizarse en modo cooperativo. Estas condiciones son las siguientes:

1. Control del acceso concurrente de múltiples usuarios al acceder al mismo objeto

2. Coherencia de los datos a manejar en la escena.

Para que se cumplan estas dos condiciones, ya hemos introducido en capítulos anteriores las suguientes técnicas:

1. Exclusión mutua entre usuarios, mediante el bloqueo de los objetos seleccionados, como se explica en el Capítulo 6.

2. Replicación de los datos de la escena en cada instancia de la aplicación, y notificación y reproducción de las modificaciones que tengan lugar sobre cada réplica.

Para que la implementación en el editor de las funciones de edición satisfaga estas dos condiciones, es necesario que la operación de edición sea aplicada únicamente a un objeto seleccionado localmente, de forma que se pueda asegurar que ningún otro usuario lo está modificando. Por otro lado, es necesario que todas las modificaciones que el usuario realice interactivamente sobre el objeto sean comunicadas al resto de réplicas presentes en la sesión cooperativa, y que esas operaciones de edición sean recreadas remotamente.

La combinación de estas dos técnicas (bloqueo previo de los objetos a editar, y envío de las modificaciones) son suficientes para que el trabajo cooperativo sea posible. En los próximos apartados, explicaremos la implementación de cada una de las operaciones básicas de edición, utilizando estas dos técnicas.

7.2. Inserción y borrado de objetos

La inserción y borrado de objetos tridimensionales son dos operaciones que modifican la estructura de la escena. Por este motivo, es necesario considerar con sumo cuidado la implementación de su soporte cooperativo, de forma que las réplicas de la escena se mantengan consistentes. El usuario puede insertar objetos en la escena desde ficheros o base de datos, o borrar cualquier objeto geométrico de la escena. A continuación explicamos la forma en que hemos implementado la estas dos operaciones en el editor cooperativo.

7.2.1. Inserción de un objeto

Cuando un objeto tridimensional es cargado desde un fichero o base de datos, como se explica en la Sección 5.4, se genera un nuevo subgrafo de escena. Este subgrafo debe ser emplazado en un lugar bajo el grafo de escena general. La posición en que se coloque variará de acuerdo con una estrategia predefinida. A continuación explicamos las dos estrategias de emplazamiento que hemos desarrollado en el editor:

- La inserción puede tener lugar en un lugar estándar del grafo de la escena. En la Sección 5.2 explicábamos como cada escena contiene un separador llamado *Objects*, bajo el cual se insertan los objetos cargados. El nuevo subgrafo de escena es añadido como hijo a este separador.

- La inserción puede tener lugar también colocando el subgrafo de escena cargado bajo un nodo de agrupación designado por el usuario. La forma de designar el lugar donde se inserta el nuevo objeto consiste en seleccionarlo primero. El nuevo objeto se añade como hijo al nodo de agrupación seleccionado.

La inserción del nuevo objeto debe ser notificada a las demás réplicas de la aplicación. Para ello se utiliza el evento ADD del protocolo Mu3D. Un evento ADD puede referirse a un fichero en disco local (clase FILE) o a una consulta a una base de datos por medio de Internet (clase URL). El mensaje a enviar contiene el nombre del fichero o URL que se debe cargar. En el momento de recibir este evento, las instancias remotas del editor lo recrean realizando la misma operación, es decir, la inserción del objeto referenciado por el evento.

7.2.2. Borrado de un objeto

El borrado de un objeto es otra de las operaciones básicas ofrecidas por el editor. El objeto seleccionado es retirado de la escena, y colocado en un espacio de almacenamiento temporal,

llamado *papelera*. A no ser que el usuario decida restaurar el objeto (como veremos en el Capítulo 9), éste permanecerá fuera de la escena.

La implementación del soporte cooperativo para la operación de borrado debe ser considerada con cuidado, ya que esta operación modifica la estructura de la escena. Si un objeto es borrado en una instancia del editor, y no en otra, las dos réplicas adquieren distinta estructura, y las rutas a los objetos quedan inconsistentes. Por tanto, una ruta que referencie a un objeto en una instancia podría no existir en otra, dando lugar a errores de programa.

Un requisito imprescindible para borrar un objeto es que ningún usuario lo tenga seleccionado en ese momento. Poder borrar un objeto seleccionado por otro usuario dificultaría la interacción en el trabajo cooperativo. Para asegurar esa condición, el programa restringe la capacidad de borrar a objetos seleccionados. De esta manera, se garantiza que ningún otro usuario lo está modificando en el momento de borrarlo.

La implementación del borrado de un objeto consiste básicamente en mover ese objeto a la papelera. Esta técnica permite que el objeto pueda ser recuperado más tarde, si es necesario.

La comunicación de la operación de borrado a las demás réplicas de la aplicación tiene lugar mediante el envío del mensaje REMOVE del protocolo Mu3D. Este mensaje contiene la ruta del objeto a borrar, de modo que al ser recibido en cada una de las máquinas que ejecutan la aplicación, el editor tiene información suficiente para recrear localmente el borrado del objeto.

7.3. Manipulación geométrica interactiva

El editor permite editar de forma interactiva los objetos geométricos que forman una escena, por medio de los *manipuladores interactivos*. En esta sección explicamos la implementación del módulo de gestión de manipuladores en el editor y su adaptación para el funcionamiento en el entorno cooperativo.

7.3.1. Introducción a los manipuladores

Un manipulador geométrico interactivo es un componente estándar de Open Inventor que proporciona al usuario la posibilidad de modificar interactivamente la matriz de transformación que afecta a un objeto.

El manipulador contiene una matriz de transformación, y es capaz de modificar el estado global de la matriz geométrica de Open Inventor cuando una acción lo atraviesa. Además, contiene geometría que funciona como interfaz de usuario, y hace posible que se pueda modificar esta matriz de transformación de forma interactiva. Esta modificación tiene lugar mediante el ratón, haciendo clic y arrastrando los diferentes controles del manipulador.

Open Inventor define distintos tipos de manipuladores. Cada uno de ellos ofrece al usuario un conjunto distinto de capacidades de interacción. Los tipos de manipuladores utilizados en la aplicación, según las facilidades de interacción que ofrecen, son los siguientes:

- El manipulador *Transformbox* permite realizar traslaciones y escalados simétricos sobre el objeto.

- El manipulador *Handlebox* permite la realización de traslaciones y de escalados asimétricos sobre el objeto.
- El manipulador *Jack* permite la realización de traslaciones, de rotaciones y de escalados asimétricos sobre el objeto.
- El manipulador *Trackball* permite realizar rotaciones de un objeto sobre su propio centro.
- El manipulador *Centerball* permite la realización de rotaciones de un objeto respecto a un punto central definible por el usuario.

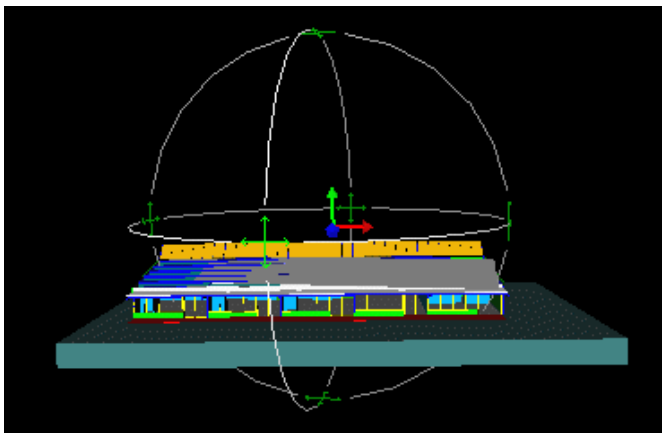

Figura 7.1.: Ejemplo de manipulador *Centerball*.

7.3.2. Gestión de los manipuladores

La clase ManipMNG

Las operaciones relativas al manejo y gestión de los manipuladores han sido encapsuladas en una clase C++, llamada *ManipMNG*. Esta clase se encarga de gestionar las operaciones básicas de manipulación interactiva sobre un objeto, que son la adición y retirada de un manipulador, y la propagación de los cambios.

Cada escena contiene su propio gestor de manipuladores, por lo que la clase ManipMNG será instanciada por cada nueva escena creada.

Adición de un manipulador

Un manipulador se asocia a un objeto por medio de una sustitución. El nodo de transformación que contiene un objeto es sustituido por el manipulador correspondiente, que se coloca en su lugar, y recibe los valores anteriores de la matriz de transformación. Esto tiene lugar por medio de una llamada al método *replaceNode* de la clase del manipulador. Un esquema de esta sustitución se puede observar en la Figura 7.2.

En caso de no existir ningún nodo de transformación en el objeto a manipular, el programa crea uno nuevo antes de sustituirlo por el manipulador, y comunica esta modificación a las demás réplicas de la aplicación mediante un evento Mu3D que contiene el evento ADD, la clase "Transform", y la ruta al objeto.

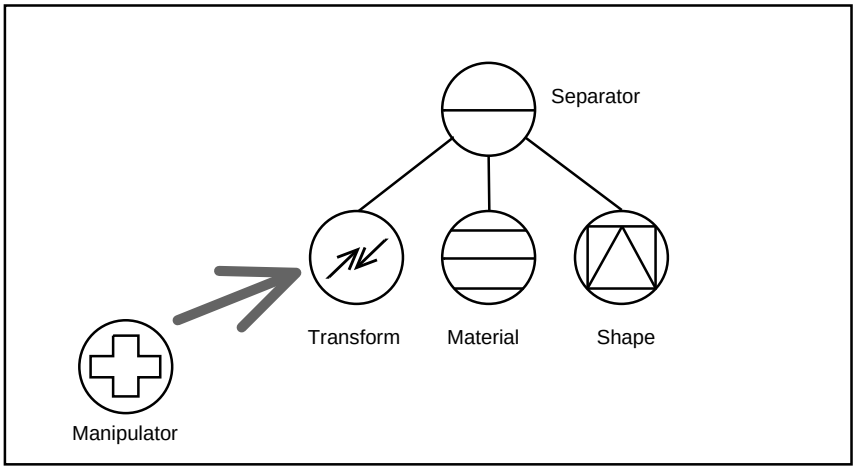

Figura 7.2.: Sustitución de un nodo de transformación por un manipulador.

Retirada de un manipulador

La retirada de un manipulador consiste en sustituir éste por un nuevo nodo de transformación, que hereda los valores de la matriz de transformación incluida en el manipulador. Esto se consigue mediante una llamada al método *replaceManip* de la clase del manipulador. Cuando el manipulador es reemplazado, es borrado implícitamente de la memoria.

Propagación de los cambios

La adición y la retirada de un manipulador son operaciones locales, que no necesitan ser comunicadas a las demás réplicas de la aplicación, ni reproducidas en éstas, ya que sólo debe poder utilizar el manipulador quien tiene seleccionado el objeto. Por tanto, estas dos operaciones no necesitan ser adaptadas para el trabajo cooperativo y no generarán el envío de ningún evento Mu3D.

En cambio, todas las modificaciones provocadas por la manipulación del objeto deben ser comunicadas a las demás instancias del editor, de manera que la matriz de transformación del objeto sea actualizada en cada réplica.

El envío de cada modificación tiene lugar dentro de una función *callback*, que es llamada cada vez que la matriz de transformación del manipulador se modifica. La ejecución de esta función envía un mensaje Mu3D de tipo MODIFY y clase "Transform", que contiene la ruta al nodo de transformación y el contenido actualizado de éste.

Cuando las réplicas remotas reciben este tipo de mensaje, sustituyen el contenido del nodo de transformación por el contenido actualizado que llega en el mensaje. De esta manera, la secuencia de operaciones seguida para manipular un objeto se ve reflejada inmediatamente en cada uno de los sitios remotos.

7.4. Materiales

Los nodos de material hacen que la escena sea visualizada mostrando sus características de color, brillo, transparencia, etc. Un usuario que esté trabajando sobre una construcción podrá distinguir más fácilmente entre distintos componentes de ésta (como por ejemplo, entre puertas y paredes), si estos componentes tienen distinto color.

En esta sección explicamos las facilidades implementadas en el editor para permitir la modificación interactiva de los materiales de un objeto en la escena. Explicaremos también la forma en que hemos implementado el soporte cooperativo para la edición de materiales.

7.4.1. Introducción a los materiales

En términos de Open Inventor, llamaremos *material* al conjunto de características que determinan los parámetros de visualización. Estas características comprenden la intensidad de cada uno de los parámetros de color, que son los siguientes:

- *Color difuso*. El color propio del objeto al que afecta el material.
- *Color ambiental*. Color reflejado de un objeto en respuesta a la luz ambiental de la escena.
- *Color especular*. Color que es reflejado por un objeto en respuesta a una iluminación directa.
- *Color de emisión*. Color de la luz emitida por un objeto.
- *Brillo*. Grado de brillantez de la superficie de un objeto.
- *Transparencia*. Grado de transparencia de la superficie de un objeto.

Todas estas características se encuentran encapsuladas en forma de campos de un nodo. Este nodo, llamado *nodo de material*, modifica el estado global de visualización, sustituyendo los valores de los atributos por los que contiene.

Open Inventor proporciona al programador un componente estándar, llamado *editor de materiales*. Este componente está implementado de forma distinta dependiendo del sistema operativo, e implementa la interfaz de usuario para la edición interactiva de un nodo de material. Como podemos ver en la Figura 7.3, el editor de materiales contiene distintos controles interactivos que permiten modificar cada uno de los campos introducidos anteriormente. La implementación es realizada en forma de una clase C++, llamada *SoWinMaterialEditor*, o *SoXtMaterialEditor*, según el sistema operativo. Además, Open Inventor proporciona una API simplificada para manejar los componentes de edición de materiales.

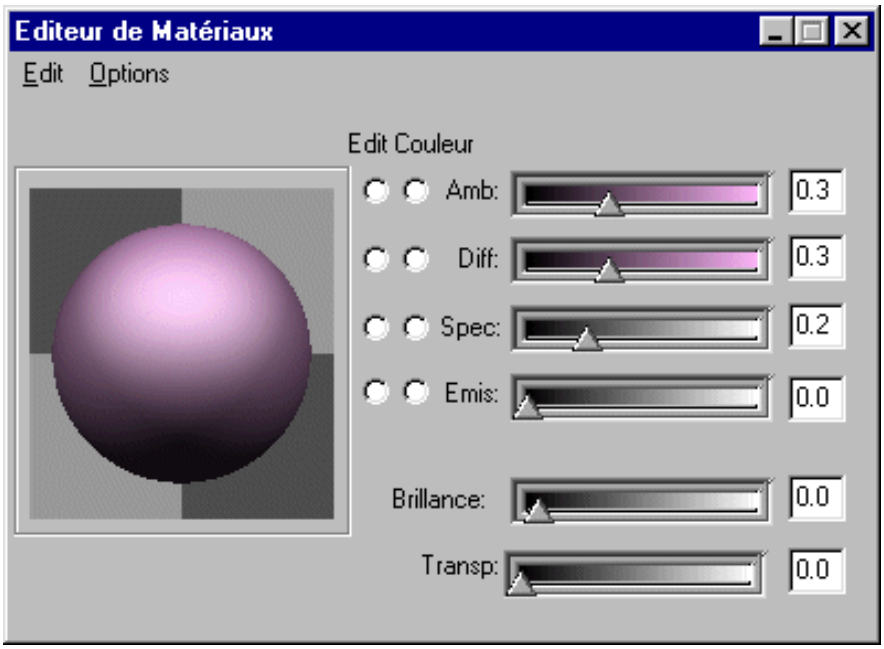

Figura 7.3.: Ejemplo de editor de materiales. Implementación para Windows NT.

7.4.2. Gestión de editores de materiales

En este apartado explicaremos la implementación de un gestor de materiales, que se encarga de realizar las operaciones necesarias, y de asegurar que se cumplen los requerimientos explicados anteriormente para el funcionamiento en forma cooperativa.

La clase MaterialMNG

Las operaciones relativas a la gestión de los editores de materiales se encuentran encapsuladas en una clase C++, llamada *MaterialMNG*. Esta clase contiene las siguientes operaciones:

- Adición de un editor de materiales.
- Retirada de un editor de materiales.
- Seguimiento de las modificaciones en el material asociado a un editor.

Existe un gestor de materiales asociado para cada escena, y por tanto, la aplicación instancia un nuevo objeto de la clase MaterialMNG para cada escena que se cree.

Adición de un editor de materiales

Para que el material de un objeto pueda ser editado con el editor de materiales, es necesario asociar el nodo de material con éste. Esto tiene lugar mediante la llamada al método *attach*

del componente. Desde el momento en que esta llamada ha tenido lugar, las modificaciones que el usuario realice en la ventana del editor de materiales se reflejarán automáticamente en los campos del nodo de material, enlazados con los controles del editor. El esquema de la asociación de un editor de materiales a un nodo de material se ilustra en la Figura 7.4.

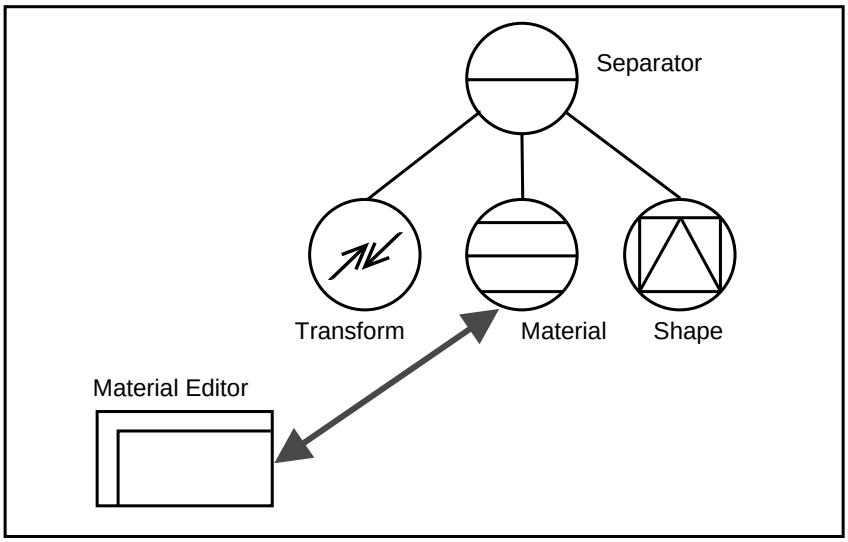

Figura 7.4.: Enlace de un editor de materiales con el nodo de material de un objeto

Si el objeto cuyo material se desea modificar no contiene ningún nodo de material, el programa inserta uno bajo el separador del objeto, y envía un mensaje Mu3D a las demás instancias, para que reproduzcan la inserción. El mensaje contendrá el evento ADD, la clase "Material" y la ruta al objeto.

Retirada de un editor de materiales

La retirada de un editor de materiales tiene lugar mediante la ruptura de la asociación de éste con el nodo de material al que estaba asociado. Esto tiene lugar por medio de una llamada al método *detach* del componente. La retirada de un editor de materiales tiene lugar en los siguientes casos:

- Cuando el usuario lo solicita explícitamente mediante una opción de menú.
- Cuando la ventana del editor se cierra.
- Cuando el objeto al que el editor de materiales estaba asociado es deseleccionado.

El editor de materiales no es borrado de la memoria al ser retirado. Un puntero a éste queda almacenado en el objeto de la clase *MaterialMNG*. Esto implica que no será necesario volverlo a instanciar para asociarlo a otro objeto.

Propagación de las modificaciones

Del mismo modo que en el caso la adición y retirada de manipuladores interactivos, la adición y retirada de un editor de materiales no generan ningún mensaje Mu3D, dado que son operaciones locales, que no deben tener repercusión en las demás instancias del editor.

En cambio, las modificaciones sobre un material deben ser comunicadas a las demás réplicas de la aplicación, de forma que el material modificado sea actualizado en todas ellas.

Los mensajes son generados por una función *callback*, que es llamada cada vez que tiene lugar la modificación del material. En ese momento, se genera un mensaje Mu3D de tipo MODIFY y clase "Material", que contiene la ruta al material a modificar, y los nuevos valores de ese material. Las réplicas del editor que reciban ese mensaje, tendrán información suficiente para ser capaces de actualizar el nodo apropiado con los nuevos valores.

7.5. Iluminación de una escena

Una escena 3D no es directamente visible si no se aplica algún tipo de iluminación sobre ella. Existen distintos algoritmos de iluminación, que producen diferentes efectos en el momento de visualizar la escena. Según el algoritmo empleado, hablaremos de distintos *tipos de luces*.

En esta sección se explica de qué manera se gestiona la iluminación de una escena en el editor, los distintos tipos de luces que se ofrecen al usuario, y las manipulaciones que éste puede realizar sobre ellas. Explicamos también la implementación del soporte cooperativo a la gestión de la iluminación, de modo que ésta se mantenga igual en todas las réplicas de la aplicación.

7.5.1. Introducción a las luces

Una luz en Open Inventor se representa como un nodo de tipo *SoLight* que se coloca en el grafo y modifica la iluminación de todos los nodos que se encuentren más a la derecha y más abajo, según los parámetros que contenga. Open Inventor define tres tipos de luces estándar:

- Las *luces direccionales* (*Directional Lights*), también llamadas *luces infinitas*, iluminan de forma uniforme en una única dirección determinada, que no depende de su posición. Un símil en el mundo real es la luz del sol.
- Las *luces puntuales* (*Point Lights*) sí ocupan una posición en el espacio 3D, desde la cual emiten luz uniformemente en todas direcciones. Son un tipo de luz similar al de una bombilla esférica.
- Las *luces focales* (*Spot Lights*) también ocupan una posición en el espacio 3D, y desde ese punto iluminan en una dirección dada. El área iluminada forma un volumen cónico. Su equivalente en el mundo real son los focos de un teatro.

Estos tres tipos de luces utilizan algoritmos de iluminación relativamente rápidos que permiten su modificación y manipulación en un entorno interactivo. Es posible asociar un manipulador a cada luz, de forma similar a los manipuladores geométricos. Es decir, el manipulador sustituye

a la luz durante el tiempo que es modificada interactivamente, y en el momento que se deja de utilizar, vuelve a ser sustituido por la luz. Un ejemplo de la interfaz de los manipuladores de luces se puede ver en la Figura 7.5.

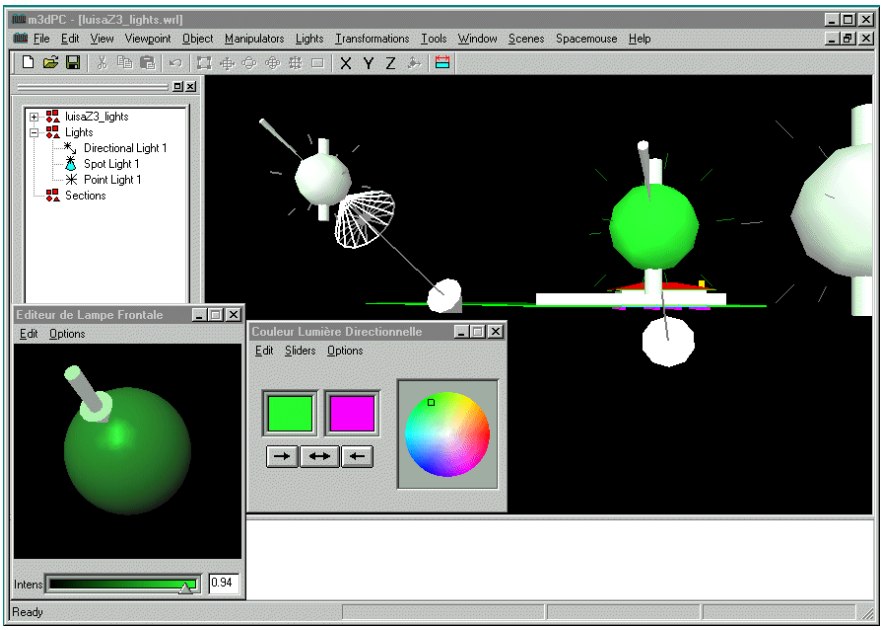

Figura 7.5.: Aspecto de una escena con tres luces añadidas.

Los tres tipos de luces presentan también diferencias en cuanto a velocidad de representación, según su complejidad. De este modo, la luz direccional es más rápida de calcular, ya que sólo implica calcular una dirección. La luz puntual implica más tiempo de cálculo, ya que se ha de calcular un punto y distintas direcciones. Por último, la luz focal es la que ocupa más tiempo de todas, ya que además se debe restringir el área iluminada.

7.5.2. Localización de las luces en cada grafo de escena

Las luces manipulables por el usuario se localizarán todas bajo el mismo subárbol, que estará a la izquierda del todo del grafo de la escena. Como se ha visto en la Sección 5.2, cada escena dispondrá de su propio grupo de luces. Este grupo está identificado por una *etiqueta*, "M3DLights", que permite reconocerlo cuando se inserta en la escena un fichero que contenga luces, de manera que todas las luces generadas por la aplicación en la escena se encontrarán juntas en el momento de editarlas.

En el momento de insertar un objeto en una escena, la aplicación busca en este objeto un subárbol que tenga la etiqueta "M3DLights". En caso de encontrarlo, coloca todos sus hijos bajo el subárbol común de luces de la escena, y después lo borra.

Cuando se debe guardar una escena en disco, la aplicación guarda junto a la geometría el subárbol de las luces, tal como se ha explicado en la Sección 5.4. Aunque la iluminación de la escena se mantenga separada de la geometría, es importante, a efectos de visualización,

que no sea volátil. Es decir, que los cambios de iluminación se puedan guardar entre sesiones cooperativas.

7.5.3. Gestión de las luces

Explicamos a continuación cómo se gestionan las operaciones sobre las luces de la escena. Estas operaciones se encuentran encapsuladas en una clase C++, e incluyen la adición de una luz, la manipulación, el borrado y la edición.

La clase LightMNG

Todas las operaciones referidas a luces, así como un conjunto de datos necesarios para su gestión, se encuentran implementadas dentro de una clase C++, llamada *LightMNG* (*Light Manager*). Existe un gestor de luces para cada una de las escenas en el editor, y por tanto esta clase se instanciará tantas veces como escenas se creen. Las acciones que la clase permite realizar sobre las luces son las siguientes:

- *Adición de una luz*. Introducción de una nueva luz en la escena.
- *Manipulación de una luz*. Modificación visual interactiva de una luz.
- *Borrado de una luz*. Eliminación de una luz de la escena.
- *Edición de una luz direccional*. Edición interactiva de una luz direccional, con la ayuda de un componente estándar.

Cada una de las acciones posibles sobre una luz se ha implementado teniendo en cuenta los aspectos cooperativos de la aplicación. En los siguientes apartados veremos las operaciones que hemos implementado para ejecutar cada una de estas acciones, así como el soporte cooperativo desarrollado para cada una de ellas.

Adición de una luz

Cuando un usuario ordena al programa añadir una luz de un determinado tipo, deben realizarse una serie de operaciones. Éstas deben garantizar que esta adición es posible, y que tiene lugar en todas las réplicas de la aplicación que se ejecutan simultáneamente. Enumeramos a continuación las operaciones que tienen lugar.

En primer lugar se comprueba que es posible añadir una nueva luz al grafo de escena. Es decir, si es posible crear un nodo más. La creación de éste tiene lugar de forma dinámica, ya que el tipo del nodo es pasado como parámetro a la función de adición de la luz.

Cuando el nuevo nodo de luz ha sido creado, es añadido como hijo al grupo de luces correspondiente a la escena adecuada. Durante la adición de las luces, la aplicación va colocando los nuevos nodos en secuencia como hijos del mismo nodo de agrupación, de forma que la última luz añadida es el último hijo del grupo de luces.

Si la adición de una luz ha tenido lugar de forma local (es decir, ha sido resultado de una orden del usuario de la instancia local de la aplicación), ésta se comunica a las demás réplicas

remotas presentes en la sesión mediante el envío de un mensaje Mu3D con el evento ADD y la clase LIGHT. Las demás réplicas ejecutarán exactamente la misma función, con la única diferencia de que no enviarán ningún mensaje por la red. El envío o no de un mensaje está parametrizado en la función de adición de la luz.

Por último, la aplicación añade un manipulador a la luz recién añadida. En el próximo apartado veremos los pasos que se sigue para añadir un manipulador.

Manipulación de una luz

Por "manipulación de una luz" entendemos el hecho de permitir al usuario modificar interactivamente los parámetros de una luz mediante un manipulador. El manipulador de una luz es un nodo, de una clase especial derivada de *SoLight*. Este nodo contiene geometría que proporciona una interfaz de usuario para manipular la luz, a la que sustituye. Un esquema de la operación de sustitución se puede ver en la Figura 7.6. En el momento de sustituir el manipulador de nuevo, los valores modificados pasan otra vez al nodo de luz original.

La adición y retirada de un manipulador de luces son, al igual que la adición y retirada de un manipulador geométrico, operaciones locales que no requieren el envío de mensajes Mu3D de inserción o borrado. No obstante, la manipulación de una luz se considera una edición, y como tal, se trata de forma análoga a la edición de cualquier nodo de geometría seleccionado. La diferencia con la selección de un nodo geométrico se encuentra en la imposibilidad de seleccionar la luz mediante una pulsación del ratón. Por este motivo, no se siguen los pasos que corresponderían a una selección estándar (explicados en el Capítulo 6), sino que se envían directamente los mensajes Mu3D de selección y deselección (con los eventos SELECT y DESELECT) en el momento de asociar un manipulador a una luz y de retirarlo.

Las modificaciones en los parámetros de una luz que resultan de su manipulación son enviadas a los demás miembros de la sesión cooperativa. Para que esto sea posible, el manipulador está asociado a una función *callback*, que es llamada cada vez que tenga lugar un cambio en alguno de los campos de la luz. Esta función *callback* se encarga de enviar un mensaje Mu3D de modificación de la luz, con el evento MODIFY y la clase LIGHT, conteniendo los nuevos valores de la luz, a todas las réplicas de la aplicación presentes en la sesión cooperativa. Cuando éstas reciban el mensaje, asignarán los nuevos valores al nodo de luz adecuado.

Borrado de una luz

El borrado de una luz equivale a eliminar el nodo asociado del grafo de escena, retirándolo del subgrafo de luces. El método encargado de esta operación se ocupa de realizar diversas comprobaciones, la más importante de las cuales se asegura de que la luz no está siendo manipulada o editada remotamente (en términos de nuestro editor, que no esté *seleccionada* por otro usuario). Esto se hace mediante una consulta a la lista de objetos seleccionados remotamente, explicada en el Capítulo 6, que también contiene las luces.

Una vez comprobado que la luz no está seleccionada remotamente, el método de borrado retira, si existen, el manipulador y el editor de luces asociado a ésta. A continuación, es enviado un mensaje Mu3D, con el evento REMOVE, la clase LIGHT y la ruta a la luz, a todas las

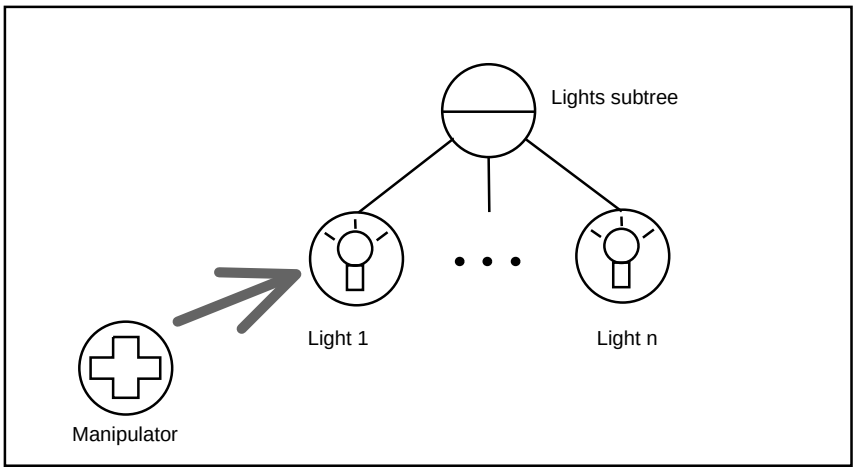

Figura 7.6.: Sustitución de una luz por un manipulador.

réplicas remotas para que eliminen a su vez la luz de sus grafos de escena. Finalmente, la luz es eliminada de la escena.

Edición de una luz direccional

Open Inventor define un componente estándar, dependiente del sistema operativo, que implementa una interfaz de usuario que permite editar interactivamente los parámetros de las luces direccionales. Este componente, llamado *editor de luces direccionales*, es implementado en forma de una clase C++, llamada *SoWinDirectionalLightEditor* o *SoXtDirectionalLightEditor*, según el sistema operativo. En el editor utilizamos este componente estándar para permitir que el usuario pueda editar la dirección, la intensidad y el color de la luz direccional de la forma que prefiera.

La adición de un editor de luces direccionales tiene lugar mediante la asociación de éste a un nodo de luz direccional. Los campos de la luz quedarán asociados a los controles de la ventana del editor, y cada modificación sobre éstos es reflejada en la escena instantáneamente. Para retirar un editor de luces direccionales, se rompe la asociación de éste con el nodo de luz.

La adición y retirada de un editor de luces direccionales no implican la inclusión o retirada de ningún nodo en las réplicas remotas de la escena, y por tanto, no se envía ningún mensaje Mu3D de inserción o retirada al resto de las réplicas.

No obstante, debemos tener en cuenta que, al igual que una manipulación, editar una luz direccional implica la necesidad de una exclusión mutua entre los usuarios, de manera que desde el momento en que se elige una luz direccional para editar, se está *seleccionando* ésta, y ningún otro usuario deberá poder seleccionarla hasta que quien la tiene la deseleccione. Por tanto, la adición y retirada de un editor de luces direccionales provocará el envío de mensajes Mu3D de selección y de deselección (con los eventos SELECT y DESELECT).

Las modificaciones producidas sobre la luz direccional asociada, mediante el uso del editor,

son capturadas por una función *callback*, que se llama cada vez que tiene lugar una modificación. Esta función se encarga de enviar los campos modificados a todas las demás instancias de la aplicación, en forma de un mensaje Mu3D que contenga el evento MODIFY, la clase LIGHT y los nuevos valores. Cuando las instancias remotas reciban este mensaje, sobreescribirán los valores del nodo de luz apropiado con los que contiene el mensaje. De esta manera, se asegura la coherencia entre las distintas réplicas de la escena en todas las instancias, también en lo que respecta a los parámetros de iluminación.

8. Portapapeles cooperativos

Las operaciones de portapapeles constituyen una metáfora muy frecuente para la transferencia de datos en las aplicaciones interactivas, y en concreto en aquellas con interfaz gráfica de usuario. Estas operaciones, llamadas *cortar*, *copiar* y *pegar*, operan sobre un espacio de memoria, llamado portapapeles, donde se pueden colocar datos de la aplicación (mediante las operaciones de cortar *y* copiar), y desde donde se pueden insertar de nuevo (mediante la operación de pegar). La mayoría de estándares de interfaces de usuario utilizan de una manera u otra esta metáfora ([IBM92],[Mic95]).

8.1. Requerimientos

La implementación de las operaciones de portapapeles no implica un problema especial en aplicaciones destinadas a un solo usuario. Su forma más sencilla tiene lugar mediante la gestión de un *buffer* de memoria. Sin embargo, en una aplicación cooperativa, es necesario que la implementación de los portapapeles cumpla una serie de condiciones:

- Por un lado, es necesario comunicar los cambios producidos por las operaciones de portapapeles en la escena a todas las demás réplicas de la aplicación, para que los reproduzcan, ya que se debe mantener la coherencia entre las múltiples copias de cada escena.
- Por otra parte, y siguiendo la filosofía básica de enviar las modificaciones en forma de mensajes lo más pequeños posible, no nos interesa enviar el contenido íntegro de los datos que se han de copiar, cortar o pegar, debido al alto tráfico de red que ello provocaría. Por otra parte, en el caso de objetos muy voluminosos, el envío de todos los datos exigiría la subdivisión de éstos en trozos, con la problemática que esto puede provocar[1].

8.2. Estructuras de datos

Siguiendo las dos condiciones introducidas en la sección anterior, hemos implementado una estructura de datos replicada para contener los portapapeles de cada usuario. La aplicación mantiene un subárbol especial destinado únicamente a los *portapapeles*, como se explica en la Sección 5.2. Este subárbol contiene un portapapeles por cada usuario presente en la sesión. De esta manera, cada instancia de la aplicación contiene los portapapeles de todos los usuarios

[1] Como por ejemplo, la interdependencia de unos mensajes respecto a otros.

en cada momento, y el contenido de éstos se debe mantener siempre coherente en todas las instancias.

Los subárboles del subárbol de portapapeles correspondientes a cada usuario reciben el nombre del *identificador de usuario* correspondiente a la sesión. De esta forma, la aplicación es capaz de acceder al portapapeles de cualquier usuario mediante el identificador de éste.

La implementación de cada una de las operaciones de portapapeles sobre esta estructura de datos viene dada en función de la tarea cooperativa a realizar.

8.3. Implementación de las operaciones

Las operaciones de portapapeles implementadas han sido las tres más usuales, es decir, cortar, copiar y pegar. Se ha desarrollado una técnica propia para las operaciones de copiar y de pegar. La implementación de la operación de cortar se ha realizado siguiendo su esquema conceptual, en forma de un operación de copiar, seguida de un borrado del objeto (de la forma explicada en la Sección 7.2).

8.3.1. Soporte cooperativo

Para la implementación de la técnica de portapapeles cooperativos, hemos implementado un nuevo tipo de evento Mu3D, llamado CLIPBOARD, que puede ser de dos clases distintas: COPY y PASTE. Cada vez que el contenido del portapapeles de un usuario cambia (es decir, cuando tiene lugar una operación de cortado, copiado o pegado), la instancia local del editor envía un mensaje Mu3D de tipo CLIPBOARD, de manera que cada una de las demás réplicas reproduzca la modificación adecuada. Los campos de este nuevo mensaje contienen información suficiente para reproducir la operación. Estos son los siguientes:

- El *identificador* del usuario que envía el mensaje.
- La *ruta* que lleva al objeto afectado.
- El *tipo de operación* a realizar (copiar o pegar).

A nivel de protocolo, no tiene sentido implementar un mensaje especial para la operación de cortar, ya que queda cubierta al subdividirla en dos operaciones atómicas: copiar y borrar. La atomicidad de la función de cortar no queda comprometida, dado que tiene lugar sobre un objeto seleccionado previamente, y por tanto, no hay peligro de que la operación de otro usuario se intercale entre las dos partes de las que consta.

Explicamos a continuación cómo hemos implementado las operaciones de copiar y pegar.

8.3.2. Implementación de la operación de copiar

La aplicación de la operación de copiar tiene lugar sobre un objeto seleccionado previamente. En rigor, no sería necesario bloquear el objeto para realizar una operación que, de hecho, no lo modifica. Sin embargo, dado que no existe en el editor ninguna otra forma de indicar el objeto

destino de una operación, y que una operación de copia puede ir seguida de un borrado (en una operación de cortar), hemos mantenido esta restricción.

Cuando un usuario quiere copiar el objeto seleccionado al portapapeles, la instancia local del programa coloca una copia en el portapapeles correspondiente al usuario local. A continuación, envía un mensaje Mu3D de tipo CLIPBOARD y de clase COPY con la ruta al objeto a copiar a las otras instancias de la aplicación en la sesión cooperativa. En el momento que una instancia remota del editor recibe un mensaje de ese tipo, accede al portapapeles correspondiente al usuario que envía el mensaje, mediante el identificador que éste contiene, y añade este objeto a este portapapeles.

8.3.3. Implementación de la operación de pegar

Cuando un usuario quiere pegar el objeto que hay en su portapapeles local en la escena, la instancia local del editor inserta en la escena activa una copia del contenido de su portapapeles. A continuación, se envía un mensaje a todos los demás miembros en la sesión cooperativa. Cuando una instancia remota del editor recibe un mensaje de tipo CLIPBOARD y de clase PASTE, accede al portapapeles que corresponde al usuario que ha enviado el mensaje y coloca una copia de su contenido en la escena activa.

El lugar donde se debe pegar el objeto está indicado en el contenido del mensaje, de forma que el usuario puede elegir en qué posición de la escena coloca el objeto. Para ello, dispone de dos opciones de menú: *Paste* (con lo que el objeto se añadirá a la escena en una localización estándar) y *Paste under selection* (con lo que el objeto se añadirá a la escena como hijo de un nodo de agrupación seleccionado).

El contenido del portapapeles no es eliminado con la operación de pegar, de forma que se puede repetir esta operación tantas veces como desee el usuario, y en los lugares de la escena que éste elija.

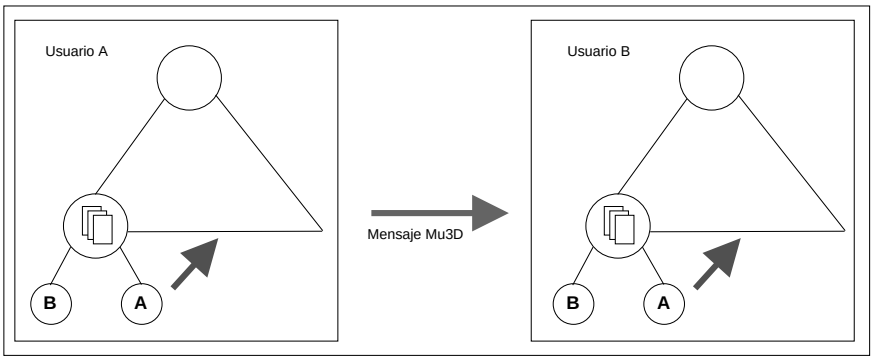

Figura 8.1.: Pegado del contenido del portapapeles del usuario A.

8.4. Conclusión

En la implementación de la técnica de portapapeles replicados que hemos desarrollado, los mensajes Mu3D enviados durante las operaciones de copiar y pegar no incluyen el contenido íntegro del portapapeles. Debido a ello, la sobrecarga de la red es mínima. De hecho, cada operación de portapapeles generará el mismo tráfico de red, independientemente del volumen de datos a transferir.

La contrapartida a esta baja ocupación del recurso de red se encuentra en la memoria extra que es necesario ocupar para mantener el subárbol de portapapeles cooperativos. Este subárbol puede contener un volumen de datos muy grande, según el tamaño de los objetos que un usuario transfiera al portapapeles, y esta ocupación extra de memoria tendrá lugar en todas las instancias de la aplicación. No obstante, dado que uno de los recursos más críticos en una aplicación distribuida y cooperativa es el ancho de banda, y que la memoria tiene un bajo coste en la actualidad, podemos afirmar que nuestra solución es satisfactoria.

El desarrollo de esta técnica de portapapeles replicados en cada instancia del editor nos permite, por tanto, mantener una funcionalidad de portapapeles que funcione de forma cooperativa, siguiendo las directrices básicas que debe cumplir toda operación en nuestro editor (mantenimiento de la coherencia en todas las réplicas), y con muy poca sobrecarga de mensajes asociada, ya que la aplicación no necesita enviar el contenido de los portapapeles, sino solamente su identificador.

9. Vuelta atrás en un entorno cooperativo

9.1. Introducción

La vuelta atrás[1] es una de las operaciones más frecuentes en cualquier aplicación interactiva moderna. Su implementación constituye, por tanto, un requerimiento para nuestro editor tridimensional, requerimiento que además ha sido solicitado por los socios usuarios del proyecto.

La implementación de la vuelta atrás en una aplicación estándar monousuario puede realizarse de forma sencilla mediante la implementación de estructuras de datos de tipo lista o pila. Estas listas pueden contener la historia de los estados de los datos, o bien la historia de modificaciones sobre los datos de usuario del programa.

Sin embargo, la vuelta atrás en un entorno cooperativo no es una tarea tan sencilla. Se debe definir un cierto "protocolo de funcionamiento", o "etiqueta", que no convierta el entorno cooperativo en un "entorno competitivo". Respecto a esta "etiqueta", hemos definido dos restricciones básicas, que se pueden enunciar de la siguiente forma:

- Un usuario no debe poder deshacer las modificaciones realizadas por otro usuario. Teniendo en cuenta que estamos en un entorno cooperativo, si un usuario deshace lo que otro ha hecho, esto puede dar lugar al efecto contrario a la *cooperación* que deseamos en nuestra aplicación.

- Además de las modificaciones ajenas, deshacer las modificaciones propias sobre un objeto no estará permitido si otro usuario lo ha modificado posteriormente. Esto nos llevaría al mismo caso que el punto anterior, ya que volver a un estado anterior a nuestras modificaciones significaría invalidar las modificaciones posteriores realizadas en la escena por otro usuario.

Además de las restricciones inherentes a la vuelta atrás cooperativa, dado que ésta tiene lugar sobre operaciones de edición, debe cumplir las mismas condiciones que el resto de operaciones cooperativas implementadas. Por tanto, cualquier cambio, ya sea hacia delante o hacia atrás, debe ser comunicado a todas las demás instancias del editor, de forma que todas las réplicas de la escena estén actualizadas.

Estas restricciones resultan relativamente severas, ya que restringen la operación de vuelta atrás a un ámbito muy concreto. Sin embargo, éstas no deben impedir que los usuarios del

[1] Llamada normalmente *Undo* en la terminología anglosajona, lo que se traduce generalmente por el verbo "deshacer". Nosotros usaremos este término cuando usemos la forma verbal, y el término "vuelta atrás" cuando utilicemos el sustantivo.

programa tengan la posibilidad de volver atrás en un momento dado, ya sea para volver a un estado anterior después de una operación de prueba, o bien para deshacer un eventual error.

En este capítulo explicamos la solución que hemos implementado, de manera que se pueda llegar a un equilibrio entre estos dos factores opuestos: por un lado, las restricciones que deben cumplirse en el entorno cooperativo, y por el otro, la posibilidad de realizar la mayor cantidad posible de operaciones de "deshacer".

Hemos de hacer notar que nuestro objetivo no ha consistido en diseñar ningún entorno general para deshacer operaciones en entornos cooperativos, al estilo de lo que se explica en [Pra92] y [Pra94]. Más bien, hemos querido desarrollar una solución concreta a un problema concreto. Esta solución, no obstante, ha sido pensada para que pueda ser extendida, como veremos más adelante.

Las operaciones correspondientes a la vuelta atrás han sido encapsuladas en una clase C++, llamada *UndoMG* (*Undo Manager*). Esta clase contiene todas las estructuras de datos definidas más adelante, y define una API para que pueda ser invocada desde el editor. Cada escena está asociada con una instancia de esta clase.

9.2. El concepto de ámbito

Para que la implementación de la operación de vuelta atrás sea posible, hemos desarrollado un nuevo concepto, que hemos denominado *ámbito*. Este concepto se define de la siguiente manera:

> *Llamaremos* ámbito *a un conjunto de operaciones sobre un objeto que pueden ser revertidas por un usuario en un momento dado.*

En un determinado momento, un usuario puede encontrarse dentro de cero o más ámbitos, y sólo será capaz de deshacer aquellas operaciones realizadas dentro de los ámbitos en los que se encuentre. Como se desprende de la definición anterior, cada ámbito lleva asociado un *objeto*, sobre el cual se han aplicado las operaciones que se pueden deshacer. Cuando el usuario sale de un ámbito, mediante una condición de salida, ya no puede volver a él.

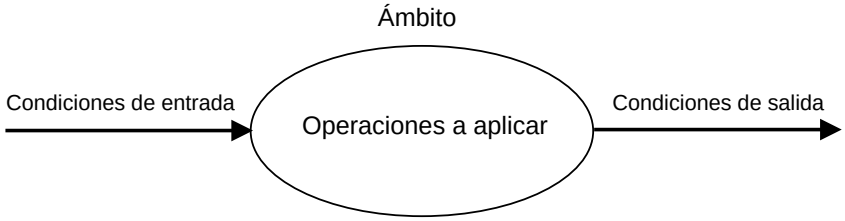

Figura 9.1.: Esquema básico de un ámbito de vuelta atrás.

Cada uno de los ámbitos tiene una serie de características propias que lo definen, en base a los cuales describiremos los distintos tipos. Un esquema que ilustra la interrelación entre las

características de un ámbito se puede observar en la Figura 9.1. Estas características son las siguientes:

- Un *conjunto de operaciones* que se pueden deshacer dentro de un ámbito sobre el objeto asociado a éste.
- Un conjunto de *condiciones de entrada*, que definen en qué momento el usuario entra en un ámbito.
- Un conjunto de *condiciones de salida*, que definen en qué momento el usuario abandona el ámbito en el que estaba.

Para definir qué ámbitos son necesarios para nuestra aplicación, hemos empezado definiendo qué operaciones se deben poder deshacer, y hemos identificado las siguientes:

- *Edición de un objeto*. Las operaciones de edición pueden comprender la modificación de su matriz de transformación geométrica, los parámetros de visualización o la modificación de alguno de sus puntos. Todas estas son operaciones que tienen lugar sobre un objeto, durante el periodo en que está seleccionado.
- *Inserción de un objeto* en la escena. Como hemos explicado en la Sección 7.2, una inserción coloca una serie de datos geométricos en la escena, bien en un lugar predeterminado, o bien bajo el objeto seleccionado.
- *Borrado de un objeto* de la escena. Como explicamos en la Sección 7.2, un borrado retira de la escena el objeto seleccionado en ese momento.

Basándonos en estos grupos de operaciones, hemos definido tres grandes ámbitos, que explicaremos a continuación. Estos son el *ámbito de selección*, el *ámbito de inserción* y el *ámbito de borrado*.

9.2.1. El ámbito de selección

El ámbito de selección se apoya en el concepto de selección, explicado en el Capítulo 6. En éste capítulo hemos explicado que un usuario es el único propietario de un objeto desde que lo selecciona hasta que realiza una deselección sobre él. Por tanto, si el usuario se encuentra dentro del ámbito de selección, el conjunto de operaciones que se podrán deshacer serán todas las realizadas desde el momento de la selección hasta el momento actual.

Definimos a continuación el ámbito de selección en términos de sus características (conjunto de operaciones, condiciones de entrada y condiciones de salida).

El conjunto de operaciones sobre un objeto seleccionado que el editor es capaz de deshacer comprende los siguientes tipos:

- Modificación de la *transformación geométrica* de un objeto, ya sea mediante un manipulador interactivo, o bien mediante un menú con datos numéricos.

- Modificación de los *parámetros de visualización* del objeto.
- Modificación de un *conjunto de vértices coplanares* del objeto.

El conjunto de condiciones de entrada se reduce a la *selección* de un objeto. El hecho de seleccionar un objeto hace que el ámbito de selección se enfoque hacia este objeto.

El conjunto de condiciones de salida está formado por las siguientes:

- La deselección del objeto asociado. Una vez que el usuario deselecciona un objeto, todas las operaciones que haya realizado sobre éste desde el momento de su selección quedan confirmadas, y no será posible deshacerlas.
- La selección de un objeto distinto al actual. Hemos visto en el Capítulo 6 como la selección de un objeto implica automáticamente la deselección del objeto que pudiera estar seleccionado anteriormente.
- El borrado del objeto asociado. En el momento de borrar el objeto asociado a la selección, todo el conjunto de operaciones realizadas sobre éste deja de estar almacenado, ya que no tiene sentido deshacerlas.

9.2.2. El ámbito de inserción

El ámbito de inserción sirve para que el usuario vuelva atrás en inserciones propias de objetos en la escena. Esto es muy útil para corregir errores que consistan en la inserción de un objeto equivocado.

El conjunto de operaciones que es posible deshacer se reduce a una sola inserción sobre el objeto asociado.

La condición de entrada en este ámbito es la inserción propia de un objeto en la escena. Esto puede tener lugar explícitamente, utilizando un comando de inserción, o bien implícitamente, mediante una operación de pegado desde el portapapeles.

El conjunto de condiciones de salida de este ámbito está formado por las siguientes:

- La inserción de otro objeto en la escena, realizada localmente. En este momento, se sale del ámbito de inserción del objeto anterior, para entrar en el ámbito de inserción de otro objeto.
- La modificación del objeto insertado. Una vez el objeto ha sido modificado, no se puede volver atrás en su inserción.
- El borrado del objeto insertado. Esto es obvio, ya que deja de existir el objeto, y por tanto su inserción no puede ser revertida.

Hemos restringido el conjunto de operaciones que se pueden deshacer a una sola inserción, por motivos de sencillez de gestión. La inserción de un objeto modifica la estructura del grafo de escena, así como los índices de las rutas de los demás nodos, con lo que extender la operativa a sucesivas inserciones puede ser complejo.

9.2.3. El ámbito de borrado

El ámbito de borrado servirá para que el usuario vuelva atrás en caso de eliminaciones accidentales de un objeto.

El conjunto de operaciones que se pueden deshacer en este ámbito se reduce únicamente al último borrado.

La condición de entrada única para este ámbito es la eliminación de la escena de un objeto cualquiera, ya sea mediante una orden explícita, o bien mediante una operación de portapapeles (la operación de Cortar).

El conjunto de condiciones de salida para este ámbito comprende las siguientes:

- La inserción de un objeto cualquiera.
- El borrado de un objeto cualquiera.

Por motivos de sencillez de gestión, hemos restringido el conjunto de operaciones al último borrado. Debemos tener en cuenta que, al igual que en el caso de la inserción, la eliminación de un objeto modifica la estructura del grafo de escena que lo contiene, así como los índices de las rutas que llevan a nodos vecinos al objeto. Estas modificaciones se complicarían más en eliminaciones sucesivas.

9.3. Almacenamiento del estado previo

Para poder deshacer una modificación, se almacena el estado anterior de la escena. En esta sección explicamos qué información es guardada, en qué estructuras, y en qué momento.

9.3.1. Almacenamiento del estado previo en el ámbito de selección

El conjunto de operaciones que se pueden deshacer sobre un objeto seleccionado comprende en la actualidad modificaciones sobre la matriz de transformación, el material y el conjunto de vértices. Para poder contemplar estos distintos tipos de operaciones y para que se puedan añadir nuevos tipos de operaciones en nuevas versiones del editor, la implementación de la vuelta atrás dentro de este ámbito se ha realizado por separado para cada tipo de operación.

La aplicación mantiene un conjunto de *listas de estado*, una por cada tipo de operación, en las cuales se almacena el estado de un objeto. La agrupación en distintas listas sigue como criterio la característica que un tipo de edición puede modificar sobre un objeto. Así, se mantiene una lista con las matrices de transformación del objeto, otra con los materiales, y otra con el subconjunto de vértices modificados, antes de producirse la modificación.

Además de mantener una lista con cada característica a modificar del objeto, se dispone de una *lista maestra*, a la que el programa consultará cuando el usuario quiera deshacer la última operación. Esta lista indica el tipo de modificaciones que se han realizado sobre el objeto en cada momento, de manera que es posible determinar de qué tipo fue la última operación, para poder acceder a la lista adecuada. Un ejemplo del estado de las listas se puede ver en la Figura 9.2.

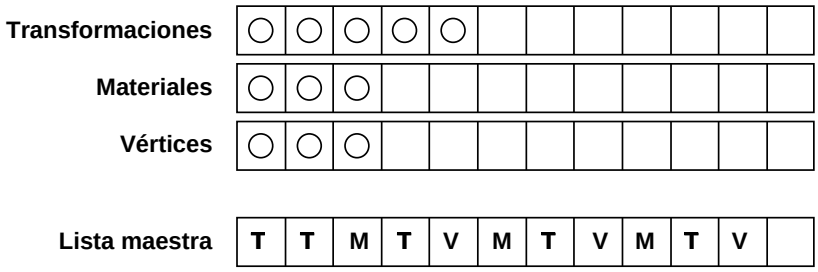

Figura 9.2.: Listas de estado utilizadas para la vuelta atrás.

Todas las listas son vaciadas cada vez que el usuario realiza una nueva selección. Cada vez que se realiza una operación atómica de edición sobre el objeto seleccionado, se añade un elemento nuevo a la lista, con el estado anterior a la operación.

Debemos definir qué consideramos una *operación atómica*, ya que esto nos indicará qué granularidad tendrá la vuelta atrás. Las operaciones atómicas que consideramos son las siguientes:

- El inicio de manipulación interactiva de un objeto. Es decir, el momento en que un usuario pulsa sobre un objeto para empezar a modificarlo.
- La modificación mediante menú numérico de algún valor de la matriz de transformación geométrica de un objeto.
- La apertura de una ventana del editor de materiales.
- La modificación de uno o más vértices del objeto.

De este modo, cada vez que el objeto es modificado, tienen lugar las siguientes operaciones sobre las listas:

- El estado previo a la modificación es añadido a la lista de estados apropiada, según el tipo de modificación.
- El tipo de modificación es añadido a la lista maestra.

9.3.2. Almacenamiento del estado previo en el ámbito de inserción

Los datos almacenados en memoria cuando el programa se encuentra en el ámbito de inserción sirven para poder deshacer la inserción realizada, y consisten básicamente en un *puntero* hacia el objeto dentro del grafo de escena, de manera que sea posible borrarlo.

Se almacena el puntero al último nodo insertado, en vez de la ruta que lleva a éste, debido a la posibilidad de que los índices de la ruta cambien, a causa de la inserción o borrado de otros nodos en el grafo de escena. Si se guarda el puntero, su valor permanece invariable aunque se modifique varias veces el grafo de escena.

Los datos son invalidados, asignándoles un valor nulo, en el momento que se abandona el ámbito de inserción.

9.3.3. Almacenamiento del estado previo en el ámbito de borrado

Los datos almacenados cuando el programa se encuentra en el ámbito de borrado permiten que el último borrado se pueda deshacer. Estos datos son los siguientes:

- Una copia del objeto borrado, situado en la *papelera* del usuario local (descrita en la Sección 5.2).
- La *ruta* hacia el nodo padre del objeto.
- La *posición* del objeto bajo el nodo padre (es decir, si era el primer hijo, el tercero, etc.).

Una copia de las papeleras de cada usuario es mantenida bajo un subárbol especial. Cada instancia del editor contiene todas las papeleras de los usuarios en la sesión cooperativa. Estas papeleras son actualizadas cada vez que tiene lugar el borrado de un objeto en cualquiera de las instancias de la aplicación:

- Cuando tiene lugar la eliminación local de un objeto, una copia de éste es colocada en la *papelera local* correspondiente a ese usuario.
- Si la eliminación del objeto es realizada por otro usuario, la copia del objeto se colocará en la papelera correspondiente al usuario que ha realizado el borrado.

En el momento de abandonar el ámbito de borrado, la papelera local es vaciada, y el resto de datos son invalidados.

9.4. Realización de la vuelta atrás

Un usuario puede deshacer la última de las operaciones realizadas en el ámbito en el que se encuentre. En esta sección explicamos los pasos a seguir en caso de realizar la vuelta atrás para cada uno de los ámbitos considerados.

9.4.1. Realización de la vuelta atrás en el ámbito de selección

Cuando un usuario selecciona el comando “Deshacer” dentro de una selección, tienen lugar las siguientes acciones:

- Consulta en la lista maestra del tipo del último cambio realizado.
- Restauración de los valores de la lista de estado apropiada dentro del objeto seleccionado, y retirada de la lista del estado antiguo.
- Retirada de la lista maestra del tipo de la última acción.

- Envío de la modificación a las demás réplicas de la aplicación en la sesión, mediante un mensaje Mu3D de tipo MODIFY, y de la clase correspondiente al tipo de modificación. Este mensaje contiene los valores a restaurar.

Mantener una lista de cada tipo de modificación realizada sobre la selección activa hace posible que se pueda ir volviendo atrás en todas las modificaciones de los tipos mencionados, hasta llegar al momento en que se ha seleccionado el objeto.

9.4.2. Realización de la vuelta atrás en el ámbito de inserción

Cuando un usuario solicita deshacer la última operación de inserción, tienen lugar los siguientes pasos:

- Selección del último objeto insertado, para evitar que sea modificado por otro usuario durante la realización de la vuelta atrás.
- Borrado de éste, y envío de un mensaje Mu3D de tipo REMOVE a las otras réplicas de la aplicación para que reproduzcan este borrado.
- Invalidación de los datos almacenados, asignándoles un valor nulo, y abandono del ámbito de inserción.

9.4.3. Realización de la vuelta atrás en el ámbito de borrado

Cuando un usuario solicita deshacer la última operación de borrado, tienen lugar los siguientes pasos:

- Restauración del objeto de la papelera local, bajo el nodo que está almacenado, en la posición adecuada.
- Envío de un mensaje Mu3D de tipo ADD y de clase RESTORE, conteniendo la ruta del objeto bajo el que se debe restaurar y la posición, a las demás réplicas de la aplicación, para que restauren el objeto de las copias locales de la papelera del usuario que envía el mensaje. En la Figura 9.3 podemos observar la restauración de un objeto del usuario A, que es replicado en el grafo de escena del usuario B. Tanto en uno como en otro caso, la papelera A es tomada como origen.
- Invalidación de los datos de lugar y posición y vaciado de la papelera local, de forma que el ámbito de borrado es abandonado.

9.5. Conclusión

La implementación cooperativa de la operación de vuelta atrás que hemos presentado cubre una gran parte del espectro completo de operaciones posibles en nuestro editor. Esto permite una funcionalidad más que aceptable, teniendo en cuenta el tipo de uso que se le dará al editor.

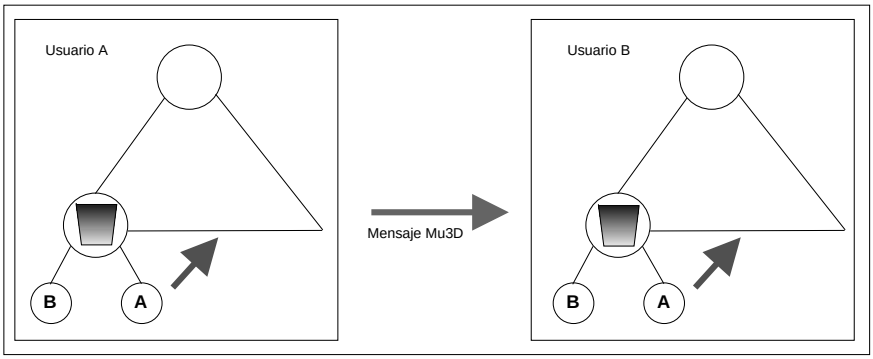

Figura 9.3.: Restauración del contenido de la papelera en una sesión cooperativa.

Por otra parte, el concepto de ámbitos ha demostrado ser extensible al implementar nuevas operaciones sobre la aplicación. En el caso de operaciones sobre un objeto, basta con añadir nuevas listas, así como una nueva rutina de gestión adaptada al tipo de propiedad que se ha de modificar. La edición de vértices de un objeto, por ejemplo, fue implementada después de definir la operación de "deshacer", y la adaptación de la operación para que pudiera ser reversible fue una tarea sencilla.

Otro ejemplo de la extensibilidad del concepto desarrollado es el siguiente. En el momento de escribir esta memoria, se está probando la implantación en el editor de un nuevo ámbito, no asociado a un objeto, relacionado con los puntos de vista de la cámara. De esta manera, el usuario será capaz de volver a una posición de visualización anterior. Esta funcionalidad fue demandada por los socios usuarios del proyecto, y su implementación fue sorprendentemente rápida y sencilla, aprovechando la estructura ya creada de la clase *UndoMG*.

La implementación de "papeleras" replicadas en cada instancia de la aplicación reduce en gran medida la sobrecarga de red, ya que no es necesario que la aplicación envíe el contenido completo de la papelera en cada momento, sino solamente su identificador. Por tanto, deshacer el borrado de un objeto provocará el mismo tráfico de red independientemente del tamaño de este objeto.

Al igual que en el caso de los portapapeles cooperativos, la contrapartida a esta baja ocupación de red se encuentra en el consumo extra de memoria que supone el almacenamiento de todas las papeleras. No obstante, la ganancia en estabilidad temporal de la aplicación (con menos retrasos provocados por saturación de la red) compensa el gasto de memoria.

Parte III.

Conclusiones

10. Conclusión

Esta memoria resume parte del trabajo que hemos realizado dentro del proyecto europeo M3D. Podemos destacar varios aspectos que han caracterizado este trabajo, y que hemos querido reflejar en esta memoria:

- En primer lugar, ha sido un trabajo de *investigación*. Se han ampliado y desarrollado estructuras de datos y algoritmos para un campo de aplicación para el que hay, todavía, terreno por explorar.
- En segundo lugar, el trabajo ha tenido mucho de *continuidad*. Nos hemos incorporado a un proyecto ya en marcha, y hemos tenido que utilizar las herramientas ya definidas, así como código que ya estaba escrito. En cierto sentido, ello puede ser más complejo que empezar a trabajar desde cero.
- Finalmente, hemos tenido que trabajar *en equipo*, de forma que nuestro trabajo y el de los otros programadores pudiera integrarse dentro de una misma aplicación. El uso de un sistema automatizado de control de versiones ha sido crucial para este trabajo en equipo.

Parte del trabajo descrito en esta memoria ha sido objeto de difusión mediante artículos publicados en congresos especializados, como se puede ver en las referencias [SC99a], [SC00a]. Una descripción exhaustiva del editor (que todavía está en desarrollo), en abril del 2000, se puede consultar en [SC00b]. Asimismo, una visión más general y de la aplicación y del sistema M3D se puede encontrar en las referencias [Gal97], [Luo98], [Gal99], [Luo99], [Luo00].

10.1. Resultados

Esta memoria ilustra la aplicación de técnicas cooperativas al caso de la edición de objetos 3D. La tarea que hemos realizado ha consistido en implementar algunas de las operaciones comunes en un editor 3D, de forma que funcionen de forma cooperativa. Los aspectos en los que nos hemos centrado han incluido:

- Aspectos de visualización.
- Aspectos de edición.

La implementación cooperativa de las operaciones cumple dos condiciones esenciales para que el trabajo cooperativo sea posible:

- Exclusión mutua entre usuarios para acceder a cada objeto de una escena.
- Coherencia continua entre las réplicas de la escena.

El conjunto de operaciones que hemos adaptado al trabajo cooperativo cubre el espectro de las operaciones más básicas sobre objetos tridimensionales.

10.2. Trabajo futuro

El tema que hemos abordado tiene un alcance muy amplio, y permite múltiples caminos para ampliar el trabajo realizado. Las posibles vías de trabajo futuro incluyen las siguientes:

- Ampliación del soporte de vuelta atrás cooperativa. Esta ampliación, demandada recientemente por los usuarios, incluye un rango más amplio de operaciones, incluidas las de visualización. Como hemos explicado en el Capítulo 9, algunas de estas ampliaciones ya han sido realizadas, y se encuentran en fase de pruebas.
- Soporte de alto nivel al trabajo cooperativo. Las técnicas cooperativas implementadas en el editor no incluyen un aspecto "de alto nivel", que ha sido requerido por los usuarios de la aplicación: el intercambio de información textual asociada a cada objeto. En la actualidad, se está trabajando en la implementación de "Post-Its", asociados a un objeto tridimensional, que cumplirán la tarea de almacenar anotaciones referentes a cada objeto. Estos "Post-Its" pueden ser utilizados para transmitir encargos entre los usuarios, durante una sesión cooperativa, y entre distintas sesiones, ya que almacenan aspectos como la fecha de emisión, responsable, prioridad, estado...
- Implementación de funciones especializadas para el trabajo arquitectónico. Más allá del alcance del trabajo presentado en esta memoria, se hallan en fase de implementación funciones cooperativas relacionadas con el trabajo arquitectónico. Estas operaciones incluyen, entre otras, las siguientes:
  - Edición interactiva de vértices de un objeto.
  - Reducción de complejidad mediante la unificación de caras coplanares de un objeto.
  - Reagrupación de los objetos de una escena, de acuerdo con el elemento arquitectónico que representan.
  - Visualización cooperativa avanzada de la escena arquitectónica (secciones y planos de corte).

Dado que el Proyecto M3D tiene una duración hasta el mes de abril del 2001, se prevé que este trabajo futuro se vea completado próximamente.

10.3. Problemas encontrados

El hecho de trabajar con librerías de alto nivel implica aceptar el funcionamiento de éstas. Este funcionamiento puede ser erróneo. En nuestro caso, dado que la librería Open Inventor es código cerrado, no ha sido posible subsanar esos errores. Otras veces, los problemas de implementación han partido de la propia naturaleza del funcionamiento de la librería, y de las herramientas utilizadas.

Como ejemplo de los problemas encontrados, podemos citar dos: el manejo de nodos VRML 97, y la gestión de la memoria dinámica en Open Inventor.

- La versión de Open Inventor utilizada contenía numerosos errores en el tratamiento de nodos VRML 97, explicados en el Capítulo 3, que han hecho imposible el manejo directo de éstos en nuestra aplicación. Por ello, hemos tenido que implementar la técnica de filtrado explicada en la Sección 5.4.

- Por otro lado, el hecho de emplear de forma extensiva memoria dinámica para la instanciación de todas las estructuras de datos, y más en un lenguaje como C++, ha obligado a que seamos extremadamente cuidadosos en las comprobaciones de punteros, siguiendo técnicas de programación segura, de forma que debíamos comprobar en todo momento si el resultado de cualquier función apuntaba a un espacio de memoria válido.

10.4. Herramientas y lenguajes

Las herramientas y lenguajes utilizados han sido determinados dentro del Proyecto M3D, y tienen relación con el entorno de desarrollo utilizado.

La versión de la aplicación para Irix ha sido implementada utilizando las siguientes herramientas de desarrollo:

- Sistema Operativo Irix 6.3 y 6.5.
- Compilador de C++ propio de Silicon Graphics.
- Las librerías gráficas utilizadas han sido las siguientes:
  - Open Inventor 2.1.2 (Silicon Graphics)
  - X Toolkit.
  - Motif.

La versión de la aplicación para Windows ha sido implementada utilizando las siguientes herramientas de desarrollo:

- Sistema Operativo Windows NT 4.0.
- Entorno de desarrollo Visual C++ 6.0, Enterprise Edition.

- Las librerías gráficas utilizadas han sido las siguientes:
  - Open Inventor 2.5.0 y 2.5.2 (Template Graphics Software)
  - Microsoft Foundation Classes (MFC).
  - Interactive Visual Framework (IVF) de Template Graphics Software.

Estas herramientas han sido ejecutadas sobre estaciones de trabajo Silicon Graphics y PC, respectivamente, equipadas con aceleración gráfica para OpenGL.

10.5. Agradecimientos

Los directores (e inspiradores) de este proyecto de fin de carrera han sido Ricardo Galli Granada y Yuhua Luo. Espero que su valiosa y complementaria ayuda para darle a esta memoria una dimensión técnica y para dotarla de coherencia se haya visto reflejada en estas páginas.

El trabajo en equipo no hubiera existido sin un equipo preparado y capaz, el equipo M3D de la UIB, formado por Toni Bennasar Obrador, Juan Fornés Chicharro, Josep Gayà Miralles, Juan Manuel Huéscar Felgueras y Juan Carlos Serra Canals. El buen ambiente, la comunicación constante y el apoyo mutuo que ha tenido lugar durante el desarrollo del proyecto ha repercutido muy positivamente en su calidad.

El trabajo de portar la aplicación de Irix a Windows NT fue desarrollado en el centro de investigación ADETTI (Lisboa, Portugal), durante una semana de marzo de 1999. Por el soporte logístico, el apoyo en el trabajo, y la exquisita hospitalidad, estoy en deuda con Miguel Dias, con Sérgio y Susana, y con Marco y Renato.

Por último, debo recordar a dos grupos de personas muy especiales. Por un lado, un reducido grupo formado por los mejores amigos y amigas que soy capaz de imaginar. Por otro, mi familia. Por su cariño, su apoyo y su paciencia incondicionales, gracias.

Parte IV.

Apéndices

A. Desarrollo de versiones concurrentes

El editor es fruto del trabajo coordinado de un equipo, formado en la actualidad por cinco personas, que trabajan concurrentemente sobre el mismo código fuente. Esta situación puede dar lugar a problemas de coordinación entre los distintos programadores en el momento de implantar una nueva funcionalidad, la solución de un fallo, o en general cualquier modificación en el código, por mínima que sea.

En concreto, nuestro equipo necesita disponer en plazos concretos de versiones estables del editor. El primer enfoque que aplicamos para obtener este resultado consistió en mantener una *copia maestra* estable del código, donde cada desarrollador aplicaría manualmente los cambios que fuera haciendo, una vez probados. Este enfoque se reveló impracticable, ya que cada desarrollador debía tener en cuenta cada uno de los cambios que había hecho en el código, sin contar los cambios ya aplicados a la copia maestra. Esto provocaba periodos de baja productividad, ya que la aplicación de un conjunto de cambios, junto a las pruebas, podía ocupar varios días de implantación de modificaciones, determinar los cambios realizados por cada programador, resolver conflictos, etc., todo ello de forma manual. Por otra parte, cada programador debía disponer de los cambios de los demás para poder continuar con su trabajo, con el consiguiente retraso.

No obstante, la necesidad de una copia maestra estable, que pueda tomarse como punto de referencia, y permita volver atrás en caso de errores en el código, es una necesidad crucial, si queremos mantener la calidad del *software* que estamos desarrollando. Esta copia maestra debe ser actualizada periódicamente, sin comprometer el tiempo de desarrollo.

Por ello, para optimizar el tiempo de desarrollo, y evitar una aplicación manual de los cambios, tarea tediosa y propensa a errores, se optó por la instalación de un sistema automático de control de versiones concurrentes, llamado CVS (*Concurrent Versions System*).

A.1. El sistema CVS

El sistema CVS ([Ced93], [Fog99]) es utilizado ampliamente en diversos entornos de desarrollo de software donde interviene un alto número de desarrolladores, y en particular en diversos proyectos de software "*open source*", donde las contribuciones de cientos de desarrolladores esporádicos están a la orden del día.

La utilidad del sistema CVS radica en la posibilidad de automatizar algunos de los procesos de modificación y comprobación de cambios en el código fuente, así como en su capacidad para guardar múltiples versiones, de forma que se pueda recuperar una versión desarrollada anteriormente en caso de ser necesario. Ello acelera enormemente el tiempo de desarrollo,

como hemos podido comprobar, ya que la implantación de cambios en el código es mucho más rápida, y la mayoría de las veces instantánea.

La elección del sistema CVS, en vez de otro sistema de control de versiones (como por ejemplo, Visual Source Safe de Microsoft, incluido con Visual C++, nuestra herramienta de desarrollo principal, y que fue también considerado) tuvo lugar teniendo en cuenta dos motivos principales:

- En primer lugar, el soporte que CVS ofrece para múltiples plataformas, y la estandarización de facto que supone su amplia implantación.
- En segundo lugar, la flexibilidad que proporciona en su uso. Otros sistemas, por ejemplo, requieren el bloqueo de los ficheros en los que está trabajando cada programador. Con CVS, en cambio, dos o más programadores pueden realizar cambios simultáneamente sobre el mismo fichero, sin necesidad de configurarlo de forma especial. Esto es importante, ya que la aplicación contiene ficheros comunes que deben ser modificados con mucha frecuencia, y a veces simultáneamente por dos o más programadores.

El sistema CVS basa su filosofía de trabajo en el mantenimiento de una base de datos centralizada, llamada *repositorio*. Este repositorio contiene el código fuente y ficheros de apoyo de los distintos proyectos software en los que se está trabajando.

Además, el sistema proporciona operaciones y mecanismos de actualización del repositorio, que permiten que la actualización de éste sea una tarea relativamente sencilla.

El sistema CVS utiliza una arquitectura cliente-servidor, que permite la interacción desde distintas máquinas clientes con el repositorio central. La parte cliente de la aplicación gestiona la conexión con el servidor central, utilizando distintos protocolos de autentificación.

Los comandos más usuales del sistema son los siguientes:

- *Checkout*. Genera una copia local de uno de los módulos del repositorio, y la vincula a éste.
- *Update*. Actualiza la copia local con el contenido actual del repositorio, avisando de qué ficheros han de ser actualizados, cuáles han sido modificados, etc.
- *Commit*. Actualiza el repositorio con el contenido modificado de la copia local.
- *Release.* Libera la copia local del vínculo con el repositorio central.
- *Add*. Añade un fichero al repositorio central. El añadido será confirmado en la siguiente ejecución del comando *commit*.
- *Remove*. Borra un fichero del repositorio central. El borrado será confirmado con la siguiente ejecución del comando *commit*.
- *Status*. Genera un listado del estado, número de versión y modificaciones de uno o más ficheros.

En el momento de actualizar la copia local del programa, mediante el comando *update*, el sistema comprueba los ficheros de código fuente modificados desde la última actualización, y los compara con los que han sido modificados localmente. En el caso de encontrar un mismo fichero modificado localmente y en el repositorio, el programa intenta combinar las dos modificaciones. Un fallo al combinar se llama *conflicto* en la terminología del CVS, y puede venir dado por dos causas:

- La combinación no ha podido iniciarse. En ese caso, el sistema guarda en el cliente una copia de seguridad de la versión local del fichero, y la sustituye por la versión del repositorio. La combinación deberá ser realizada a mano por el programador.
- La combinación se ha iniciado, pero no ha podido llevarse a cabo correctamente. En este caso, el sistema guarda en el cliente una copia de seguridad de la versión local del fichero, y la sustituye por uno nuevo, que contiene las secciones de código que entran en conflicto, delimitadas por caracteres especiales. El programador deberá resolver manualmente estos conflictos.

Además de los comandos anteriores, que contienen múltiples opciones, el sistema CVS proporciona otros, que permiten identificar versiones distintas de un módulo de software. Nosotros hemos utilizado únicamente el comando *Tag*, que asigna etiquetas a distintas versiones. Estas versiones pueden recuperarse más tarde desde el repositorio, identificándolas por su etiqueta.

A.2. Política de utilización

Las operaciones que hemos descrito en la sección anterior no son suficientes para impedir que las distintas versiones del programa pierdan la coherencia, o que un desarrollador sobreescriba los cambios de otro por error. Como se explica en el manual de uso de CVS ([Ced93]), es necesario definir y aplicar una *política de utilización* del sistema, que permita mantener un nivel de productividad y de calidad elevado.

A continuación explicamos las reglas de uso que hemos definido para la correcta utilización del sistema CVS en el desarrollo del editor.

- Actualización frecuente de las copias locales desde el repositorio, de forma que cada programador tenga la versión del código fuente lo más actualizada posible. Con ello, se minimiza la presencia de conflictos, y se favorece su rápida detección y solución.
- Antes de realizar una actualización de la copia local (con el comando *update*), es muy útil simular esa actualización, utilizando el comando *-n -q update*. Este comando muestra los mismos mensajes de salida que *update*, sin realizar ninguna de las modificaciones de éste. De este modo, se puede comprobar qué ficheros han sido modificados, tanto en la copia local como en el repositorio, y los posibles conflictos pueden ser detectados antes de que aparezcan.
- Antes de enviar los cambios de la copia local al repositorio, es obligatorio realizar una actualización de ésta, y resolver los conflictos que aparezcan. Hemos de hacer notar

que el sistema CVS puede detectar los conflictos, y marcarlos, pero no garantizará la integridad del código, ni prohibirá que se envíen cambios que contengan conflictos.

- Por la misma razón, se intenta no enviar al repositorio una versión que no haya sido probada suficientemente con anterioridad. Si un programador envía al repositorio una versión inestable, o que incluso no se pueda compilar correctamente, propagará estos errores al resto del equipo, sin que el sistema CVS pueda evitarlo.

Esta política de utilización del sistema CVS ha sido difundida entre el equipo de programadores, y aplicada al desarrollo del editor cooperativo. Su aplicación nos ha permitido mantener un tiempo de desarrollo sostenido, evitando retrasos por motivos de integración.

A.3. Configuración y aplicación

La instalación de un sistema cliente-servidor debe adaptarse al entorno *hardware* existente, y a las necesidades concretas de funcionamiento de este entorno. En esta sección explicamos cómo se ha instalado el sistema CVS en las plataformas de que dispone el laboratorio, para hacer posible su utilización durante el desarrollo del editor. Además, describiremos cómo se ha realizado la integración de los clientes CVS con los entornos de desarrollo, y qué herramientas de soporte han sido añadidas al sistema.

La estructura cliente-servidor del sistema CVS se ha implantado en las máquinas del laboratorio de la siguiente forma:

- El servidor se ha instalado en el servidor general del proyecto M3D. Este servidor es un Intel Pentium III, con el sistema operativo Linux Red Hat 6.1.

- Los clientes han sido instalados en estaciones gráficas PCs con Windows NT, y en una estación de trabajo Silicon Graphics con el sistema operativo Irix 6.5.

Un esquema de esta estructura se puede observar en la Figura A.1.

La ejecución de la parte cliente del sistema tiene lugar mediante línea de comandos. Los argumentos del comando CVS pueden ser introducidos de forma textual, mediante una interfaz de comandos. Esta interfaz textual, no obstante, ha sido integrada con el entorno de desarrollo de Visual C++, en forma de herramienta externa. De esta forma, los parámetros pueden ser introducidos desde el entorno integrado, y la salida textual del comando es redirigida a una de las ventanas de texto del entorno Visual C++.

La consulta del repositorio tiene lugar, por lo general, mediante los comandos *log* y *status* del sistema CVS. La salida de este comando es generada en forma textual, de forma que su consulta es poco amigable. Para mejorar este aspecto, hemos instalado herramientas gráficas de consulta al repositorio. Estas herramientas presentan una salida hipertextual a los comandos introducidos.

La herramienta CVS2HTML [Tof] es un programa que se ejecuta periódicamente sobre el repositorio, y genera un conjunto de páginas web estáticas que contienen el estado de cada

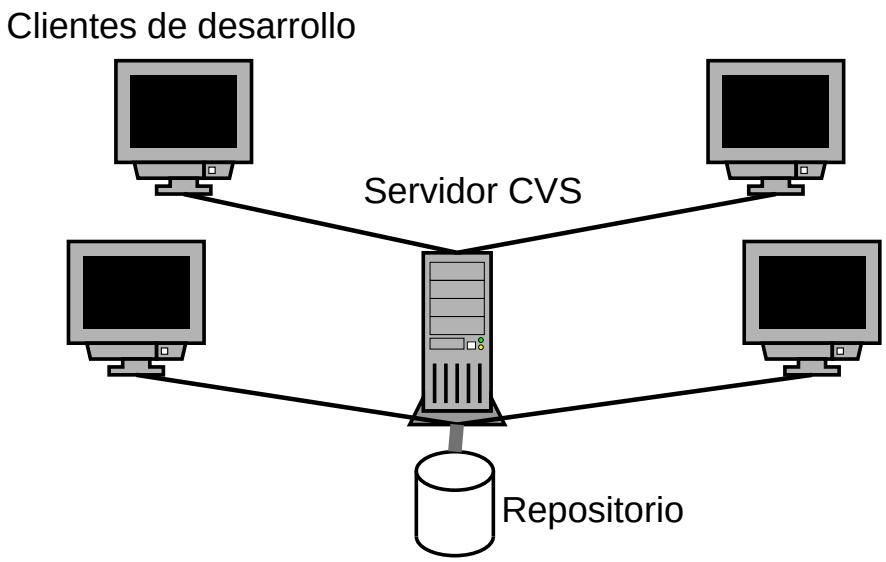

Figura A.1.: Esquema de la instalación del CVS en el laboratorio M3D de la UIB.

fichero de código fuente. Un aspecto de la interficie de esta herramienta, que ofrece una imagen general de cada fichero en el contexto de los directorios, se puede ver en la Figura A.2.

La herramienta CVSWeb [Fen], en cambio, es un *script* CGI que se ejecuta cada vez que el usuario accede a la página web. Ello permite un acceso permanentemente actualizado al estado de todos los ficheros del repositorio, ya que las páginas son generadas dinámicamente a partir del estado de cada fichero. El acceso a la historia de cambios se realiza en base a los distintos ficheros de código fuente, de forma que es posible consultar qué cambios han sido realizados últimamente en un fichero concreto, y por quién, como se puede ver en la Figura A.3.

Las herramientas de acceso web añaden a la potencia del sistema CVS la facilidad de consulta de los cambios, de manera que el conjunto de los ficheros de código es fácilmente controlable. Esto es indispensable en un proyecto como el nuestro, donde el volumen de código es considerable (unas 70000 líneas de código en la actualidad).

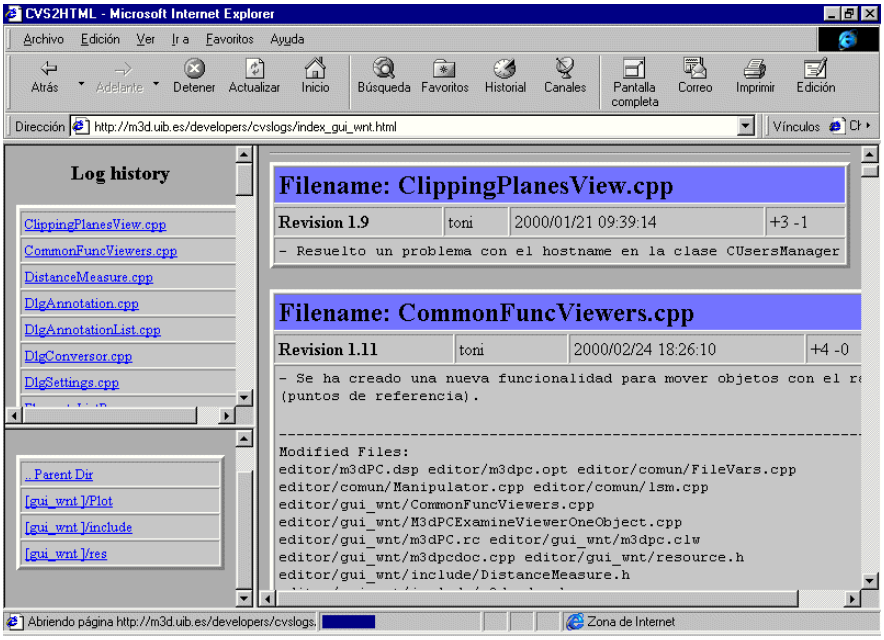

Figura A.2.: Aspecto de la interfaz de CVS2HTML

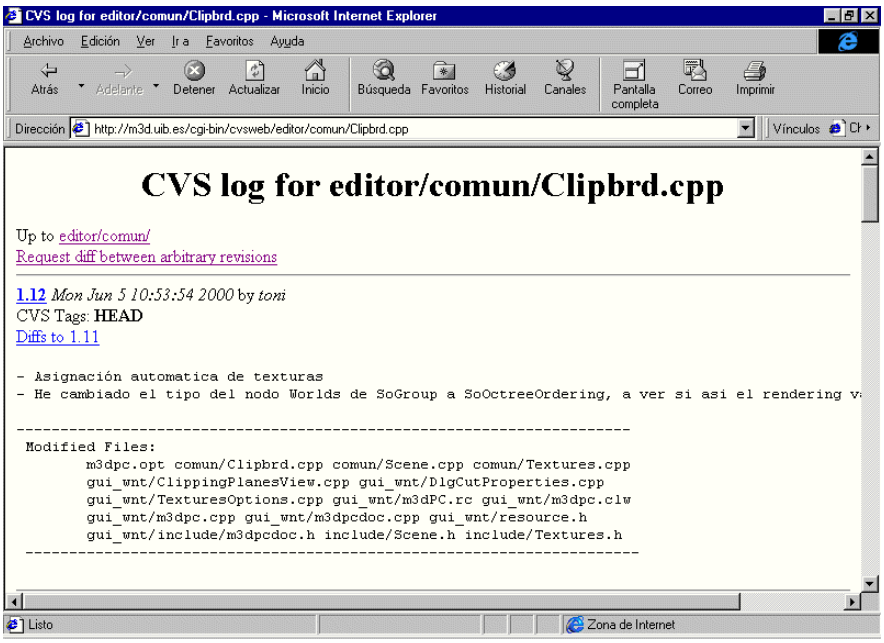

Figura A.3.: Aspecto de la interfaz de CVSWeb

B. Portabilidad y múltiples plataformas

B.1. Consideraciones previas

Uno de los requisitos de nuestra aplicación desde el momento de la definición del proyecto es que sea *multiplataforma*. Es decir, que pueda funcionar sobre distintos sistemas operativos, conservando una funcionalidad lo más similar posible entre todos ellos. Los dos sistemas operativos requeridos en el proyecto M3D han sido Irix de Silicon Graphics y Windows NT de Microsoft, sobre los que hablaremos más adelante.

El hecho de utilizar un conjunto de librerías estándar como es Open Inventor nos ha permitido construir la aplicación sobre distintas plataformas, minimizando el tiempo de desarrollo dedicado a implementar funciones diferentes para cada plataforma. Dado que la mayor parte de la aplicación utiliza la API de Open Inventor, portarla a otras plataformas requiere tener en cuenta dos aspectos:

- Que la plataforma elegida cuente con una versión de Open Inventor (lo que ocurre con la mayor parte de sistemas Unix, y con las últimas versiones de Microsoft Windows).
- Que los aspectos de interfaz de usuario sean considerados de forma separada respecto a las funciones básicas de la aplicación.

La implementación de la interfaz de usuario es muy distinta según el sistema operativo y las librerías a utilizar, ya que cambia el paradigma de programación. El hecho de utilizar Open Inventor nos ha condicionado, por otra parte, a escoger uno u otro paradigma de programación de interfaces gráficas de usuario, ya que la versión de Open Inventor para Unix se encuentra muy ligada al *X toolkit* ([Rei92], [Nye92]), mientras que la versión para Microsoft Windows ofrece facilidades para que se la use conjuntamente con las *Microsoft Foundation Classes* o MFC ([Tem97]). Utilizar Open Inventor para Windows sin ligarlo a las MFC supondría renunciar a una reducción en el tiempo de desarrollo de la interfaz de usuario, lo cual no es admisible en un proyecto tan complejo como el nuestro, y con plazos de entrega ya fijados para la aplicación.

Para facilitar la generación de código portable, se ha enfatizado la separación en distintos *módulos*, con un nivel de acoplamiento relativamente bajo, de forma que cambiar uno implique el mínimo número de cambios en los demás.

Es posible que en un mismo módulo se deban separar grupos de instrucciones de código, según la plataforma a utilizar. Para ello, utilizamos *directivas de precompilación*, que indican al compilador qué código reconocer, según el sistema operativo. De esta manera, el código ejecutable generado será distinto para cada una de las plataformas consideradas.

B.2. Portar la aplicación de Irix a Windows NT

En esta sección explicamos el proceso seguido para portar la aplicación a partir de su estado inicial, en sistema operativo Irix, a un código capaz de ser compilado y ejecutado correctamente tanto en Irix como en Windows NT. Este proceso tuvo lugar durante una semana de trabajo en la sede de ADETTI (Lisboa), otro de los socios desarrolladores del proyecto M3D, con cuyo equipo de desarrollo colaboramos en la adaptación para la portabilidad.

B.2.1. Estado inicial de la aplicación. La plataforma Irix

La aplicación se empezó a desarrollar únicamente sobre el sistema operativo Irix, de Silicon Graphics. Se desarrolló una interfaz de usuario muy sencilla, con una única ventana de visualización. Los comandos de usuario eran introducidos mediante un menú. El aspecto inicial de la aplicación se puede observar en la Figura B.1.

La interfaz de usuario se desarrolló basándose en el sistema de ventanas X-Windows, utilizando la librería *X toolkit* ([Rei92], [Nye92]). Esta librería define el conjunto de operaciones básico para gestionar los elementos gráficos de control, a los que llama *widgets*. Para poder ser utilizada, necesita asociarse a un conjunto separado de *widgets*. La versión para Irix de nuestra aplicación utiliza el conjunto de *widgets* Motif ([Hel94]), uno de los más estandarizados en entornos Unix, sobre el cual se implementa una gran cantidad de aplicaciones.

La versión de la aplicación de la que partimos contenía un conjunto mínimo de funciones. La implementación de algunas de ellas era incompleta, y otras se encontraban en fase de pruebas. Estas funciones incluían:

- Navegación a lo largo de una única escena.
- Selecciones básicas.
- Manipulación geométrica interactiva.
- Edición de materiales.
- Gestión de iluminación.

Tomamos la decisión de portar esta versión inicial de la aplicación al sistema operativo Windows NT para continuar el trabajo sobre esta plataforma por distintos motivos, que incluyen:

- El sistema operativo Windows NT era una de las plataformas de ejecución requeridas por el proyecto M3D.
- El bajo coste de las estaciones gráficas de desarrollo basadas en PC, frente a las basadas en la arquitectura Silicon Graphics hacía más sencillo obtener el equipamiento necesario para el desarrollo.
- La mayor facilidad de obtener para la plataforma Windows NT herramientas de desarrollo con entornos integrados, y capaces de generar interfaces de usuario visualmente. En nuestro caso, el entorno Visual C++ 6.0.

La necesidad de portar la aplicación en ese momento vino dictada por la complejidad creciente de la aplicación, que hacía prever que portarla más adelante resultaría más complicado.

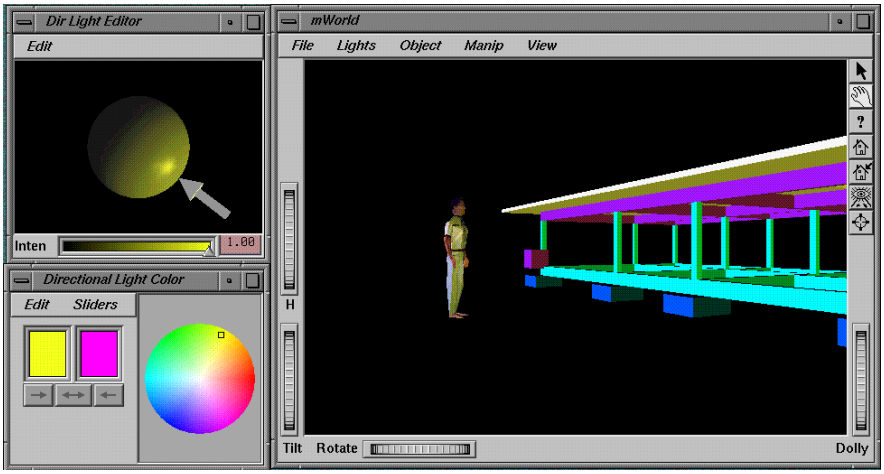

Figura B.1.: Aspecto de la implementación de la aplicación para el sistema operativo Irix.

B.2.2. Windows NT y el entorno de programación Visual C++

Windows NT es un sistema operativo propiedad de Microsoft, que define una API completa de funciones. Éstas permiten escribir aplicaciones "desde abajo", siguiendo unas líneas de estilo definidas por Microsoft ([Mic95]). Para reducir la complejidad y la lentitud de desarrollo que ello supone, distintos fabricantes han creado librerías de clases C++, que se ajustan a estas líneas de estilo, y que permitan al programador trabajar a más alto nivel, sobre todo en lo que respecta a los detalles de interfaz de usuario, con la consiguiente reducción del tiempo de desarrollo.

El propio Microsoft desarrolla y distribuye una librería de clases de este tipo, llamada *Microsoft Foundation Classes* o MFC ([Feu97], [Kai99]), que viene integrada con su entorno de desarrollo, Visual C++. Esta fue la librería elegida para el desarrollo del editor, ya que la versión para Windows NT de Open Inventor está basada en este entorno, y además presenta facilidades para la integración con MFC.

La librería MFC favorece el desarrollo de aplicaciones que sigan el modelo documento-vista. Este modelo no es más que una aplicación, con otro nombre, del paradigma *Model-View-Controller,* o MVC ([Lew95]), desarrollado en Xerox Parc junto con el lenguaje orientado a objetos Smalltalk. Explicamos a continuación en qué consiste el modelo documento-vista.

El modelo documento-vista separa los datos de la aplicación (llamados *documento* en la terminología de Microsoft) de la presentación que se da a éstos (a esta presentación se la llama *vista*). Los datos de un mismo documento pueden estar representados por numerosas vistas diferentes. Por ejemplo, una hoja de cálculo puede presentar los mismos datos en dos ventanas distintas, una en forma de celdas, y la otra en forma de un gráfico de barras.

La interacción con el usuario es controlada por la ventana principal de la aplicación, llamada *Frame Window* en terminología de MFC. Esta ventana recoge y procesa los eventos de la interfaz de usuario y los pasa al documento o a la vista.

La tripleta formada por el documento, la vista y ventana principal es coordinada y controlada por una cuarta clase C++, que representa la aplicación. Esta clase es responsable de realizar todo el proceso de inicialización, ejecución y recepción de mensajes necesario en una aplicación para Windows.

Los mensajes de Windows son tratados por MFC mediante manejadores de mensajes (*handlers*) asociados a cada clase principal. Estos manejadores son funciones *callback* que se ejecutan en el momento en que llega el mensaje asociado. Los manejadores son asociados con los mensajes correspondientes mediante un mapa de mensajes (*message map*).

Este modelo de desarrollo de aplicaciones es favorecido en gran medida por las MFC, así como por el entorno de desarrollo Visual C++. Por tanto, una aplicación que quiera aprovechar el potencial que el entorno de desarrollo ofrece (aspectos de interfaz de usuario, generación automática de código para las clases C++, etc.), deberá ceñirse a este modelo de desarrollo.

B.2.3. El proceso de adaptación de código

La adaptación de código para portar la aplicación de Irix a Windows NT siguió un proceso durante el cual se identificaron las porciones de código comunes a ambas plataformas, y las que eran dependientes del sistema operativo empleado.

De esta forma, los módulos que contenían únicamente código estándar de Open Inventor, y que constituían el corazón de la aplicación, fueron aislados. Al ser estos módulos considerados independientes de la plataforma, los módulos dedicados a la interfaz de usuario realizarían llamadas a los módulos comunes.

A continuación, se realizó un proceso de identificación de los elementos principales de la aplicación con los elementos principales de una aplicación MFC. Tras este proceso, obtuvimos los siguientes resultados:

- El grafo principal de cada escena se relacionó con la clase correspondiente al documento, *CM3dpcDoc*.
- Cada uno de los visores de Open Inventor utilizados se relacionó con las distintas vistas de la aplicación, como la clase *CM3dpcView*.
- Para la ventana principal de la aplicación se creó una nueva clase, *CMainFrame*.
- Por último, el código correspondiente a la inicialización de la aplicación se situó en el método apropiado de la clase aplicación. La clase se llama *CM3dpc*, y su método de inicialización estándar es *OnInitApp()*.

Una vez realizado este proceso de identificación, se reescribieron las operaciones correspondientes al árbol de menús, y la barra de botones, de forma que pudimos obtener una aplicación ejecutable en Windows con una funcionalidad equivalente a la anterior versión, que funcionaba en Irix.

B.2.4. Proceso de pruebas y problemas encontrados

El proceso de pruebas seguido para comprobar los resultados consistió en una prueba exhaustiva de todas las funciones existentes en aquel momento en la aplicación.

Las pruebas se realizaron en modo cooperativo entre dos máquinas, una estación de trabajo Silicon Graphics y un PC. Estas pruebas permitieron identificar problemas y puntos a tener en cuenta.

El principal problema encontrado en esta fase tuvo que ver con las diferentes formas de representar las rutas de los directorios en los distintos sistemas operativos. Como vimos en el Capítulo 4, algunos de los mensajes Mu3D envían la ruta completa de un archivo. Los distintos caracteres utilizados por una y otra plataforma ("/" en Unix, y "\" en Windows), producían problemas de ejecución en las dos plataformas, por lo que tuvieron que ser sustituidos por una representación neutra (el carácter "|"). Esta representación neutra es convertida al carácter adecuado, según la plataforma.

B.3. Desarrollo simultáneo de la aplicación en Irix y Windows NT

Una vez se dispone de una misma versión de la aplicación para Windows NT, es necesario continuar su desarrollo simultáneamente en ambas plataformas. Para ello, hemos seguido dos técnicas principales, que explicamos a continuación.

En primer lugar, hemos organizado el conjunto de ficheros fuente en directorios, de forma que el conjunto de instrucciones de compilación en cada plataforma (*Makefile* en Irix, archivo de proyecto en Windows) incluya únicamente los ficheros necesarios. La estructura principal de los ficheros es la siguiente:

comun Directorio que contiene los ficheros de implementación comunes a las dos versiones de la aplicación. Se compone principalmente de código estándar en C++ basado en Open Inventor.

include Directorio con los ficheros de cabecera comunes a las dos versiones de la aplicación.

gui_wnt Directorio con los ficheros de implementación propios de la interfaz de usuario de Windows NT. Se compone principalmente de código en C++ utilizando MFC.

- **include** Directorio con los ficheros de cabecera propios de la interfaz de usuario de Windows NT.

gui_sgi Directorio con los ficheros de implementación propios de la interfaz de usuario de Irix. Se compone principalmente de código basado en X Toolkit y Motif.

- **include** Directorio con los ficheros de cabecera propios de la interfaz de usuario de Irix.

En segundo lugar, hemos introducido, en los ficheros de código fuente comunes que lo necesitaban, directivas de preprocesado, que permiten que se genere uno u otro código según la directiva. Un ejemplo de esta técnica se puede ver en la Figura B.2.

Esta técnica se ha aplicado, sobre todo, en funciones cuya implementación deba ser distinta para las diferentes plataformas.

```
void myFunction(void) {
  #ifdef WIN32
    // Fragmento de código para Windows
  #elif IRIX53__
    // Fragmento de código para Irix
  #endif
}
```

Figura B.2.: Ejemplo de separación de código según la plataforma.

La combinación de estas dos técnicas nos ha permitido obtener dos resultados diferentes. Por un lado, mantener el código independiente de la plataforma separado del código dedicado a la interfaz de usuario. Por el otro, ha sido posible realizar dos implementaciones alternativas de la misma función cuando éstas eran distintas para las diferentes plataformas.

Parte V.

Índices y bibliografía

Índice de Figuras

Bibliografía

[Aho88] Alfred V. Aho, John E. Hopcroft, y Jeffrey D. Ullman. *Estructuras de datos y algoritmos*. Addison Wesley Iberoamericana, 1988.

[Alm95] A. Almeida y C. A. Belo. Support For Multimedia Cooperative Sessions Over Distributed Environments. En Southampton Society for Computer Simulation, editor, *Proceedings Mediacomm '95*. abril 1995.

[Aut90] AutoDesk, Inc., San Rafael, California. *AutoCAD Reference Manual*, 1990.

[Aut94] AutoDesk, Inc., San Rafael, California. *3D Studio Release 3 File Toolkit: Reference Manual*, junio 1994.

[Bro96a] Wolfang Broll. Extending VRML to support Collaborative Virtual Environments. En UK University of Nottingham, editor, *Proceedings of CVE 96, Workshop on Collaborative Virtual Environments*. septiembre 1996.

[Bro96b] Wolfgang Broll. Interaction and Behaviour in Web Based Shared Virtual Environments. En *Proceedings of the IEEE Global Internet 96*. London, noviembre 1996.

[Bro97a] Wolfgang Broll. Distributed Virtual Reality for Everyone: A Framework for Networked VR on the Internet. En IEEE Computer Society Press, editor, *Proceedings of the IEEE Virtual Reality Annual International Symposium 1997 (VRAIS 97)*, páginas 121–128. 1997.

[Bro97b] Wolfgang Broll. Populating the Internet: Supporting Multiple Users and Shared Applications with VRML. En ACM SIGGRAPH, editor, *Proceeding of the VRML 97 Symposium*, páginas 87–94. Monterrey, CA, febrero 1997.

[Bud94] T. Budd. *Programación Orientada a Objetos*. Addison Wesley, 1994.

[Cad] CadStudio. VRMLout for AutoCAD. http://www.cadstudio.cz/indexuk.html.

[Car97] Rikk Carey, Gavin Bell, y Chris Marrin. ISO/IEC 14772-1:1997, Virtual Reality Modeling Language, (VRML97), 1997. http://www.vrml.org/Specifications/VRML97.

[Ced93] Per Cederqvist. *Version Management with CVS*. Signum Support AB, 1993.

[Fen] Bill Fenner, Henner Zeller, Henrik Nordström, y Ken Coar. CVSWeb. http://stud.fh-heilbronn.de/zeller/cgi/cvsweb.cgi.

[Feu97] Alan R. Feuer. *MFC Programming*. Addison Wesley, Reading, Mass., 1997.

[Fog99] Karl Fogel. *Open Source Development with CVS*. The Coriolis Group, octubre 1999.

[Fol92] J. D. Foley, A. Van Dam, S. K. Feiner, y J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison Wesley, Reading, Massachusetts, second edición, 1992.

[Gal97] R. Galli, P. Palmer, M. Mascaró, M. Dias, y Y. Luo. A Cooperative 3D Design System. En *Proceedings of CEIG97, Barcelona, Spain*. junio 1997.

[Gal99] Ricardo Galli, Yuhua Luo, David Sánchez, Sérgio Alves, Miguel Dias, Renato Marques, Antonio Almeida, Jorge Silva, José Manuel Fonseca, y Bas Tummers. M3D Technical Specifications. Deliverable 1.2, ESPRIT Project No 26287 M3D, abril 1999.

[Gal00] Ricardo Galli y Yuhua Luo. Mu3D: A Causal Consistency Protocol for a Collaborative VRML Editor. En ACM SIGGRAPH, editor, *Proceeding of the VRML 2000 Symposium*. Monterrey, California, febrero 2000.

[Har96] J. Hard y J. Wernecke. *The VRML 2.0 Handbook*. Addison Wesley Publishing Company, 1996.

[Hea94] D. Hearn y M. P. Baker. *Computer Graphics*. Prentice Hall International Editions, second edición, 1994.

[Hec97] Michael M. Heck. VRML 2.0 for Open Inventor Programmers. A Technical White Paper. Informe técnico, Template Graphics Software, 1997.

[Hel94] Dan Heller, Paula M. Ferguson, y David Brennan. *Motif Programming Manual*, tomo 6A de *The Definitive Guides to the X Window System*. O'Reilly and Associates, febrero 1994.

[HF99] Juan Manuel Huéscar Felgueras, Ricardo Galli, y Yuhua Luo. Conversión de objetos CAD en VRML. En *Actas del IX Congreso Español de Informática Gráfica*. junio 1999.

[HO93] Enrique Hernández Orallo y José Hernández Orallo. *Programación en C++*. Editorial Paraninfo, 1993.

[IBM92] IBM. *Object-Oriented Interface Design: IBM Common User Access Guidelines*. IBM, Carmel, Indiana, Que., 1992.

[Kai99] Eugène Kain. *The MFC answer book: Solutions for effective Visual C++ applications*. Addison Wesley, Reading, Mass., 1999.

[Lew95] Simon Lewis. *The Art and Science of Smalltalk*. Prentice Hall / Hewlett-Packard Professional Books, 1995.

[Lip91] Stanley B. Lippman. *C++ Primer*. Addison Wesley Publishing Company, 2a edición, diciembre 1991.

[Luo98] Y. Luo, R. Galli, M. Mascaró, y P. Palmer. Cooperative Design For 3D Virtual Scenes. En *Proceedings of the Third IEEE International Foundation On Cooperative Information Systems (CoopIS'98), New York, USA*, páginas 373–381. agosto 1998.

[Luo99] Y. Luo, R. Galli, A. Almeida, y M. Dias. A Prototype System For Cooperative Architecture Design. En *Proceedings of IEEE Symposium on Information Visualization*. julio 1999.

[Luo00] Yuhua Luo, Ricardo Galli, David Sánchez, Antonio Bennasar, Antonio Carlos Almeida, y Miguel Dias. A Cooperative Architecture Design System Via Communication Networks. En Stanford University, editor, *Proceedings of the 8th International Conference On Computer Aided Design, Building Construction and Civil Engineering*. agosto 2000.

[Mic95] Microsoft. *The Windows Interface Guidelines for Software Design*. Microsoft Press, 1995.

[Nye92] Adrian Nye y Tim O'Reilly. *X Toolkit Intrinsics Programming Manual for X11, Release 5*, tomo 4 de *Definitive Guides to the X Window System*. O'Reilly and Associates, agosto 1992.

[Pra92] Atul Prakash y Michael J. Knister. Undoing Actions In Collaborative Work. En *Proceedings Of ACM CSCW'92 Conference On Computer-Supported Cooperative Work*, páginas 273–280. octubre 1992.

[Pra94] Atul Prakash y Michael J. Knister. A Framework For Undoing Actions In Collaborative Systems. *ACM Transactions on Computer-Human Interaction*, 1(4):295–330, 1994.

[Rei92] Tim O Reilly, Mark Langley, y David Flanagan (Editor). *X Toolkit Intrinsics Reference Manual for Version 11 of the Window System*, tomo 5 de *Definitive Guides to the X Window System*. O'Reilly and Associates, 3a edición, julio 1992.

[Rul96] Keith Rule. *3D Graphics File Formats. A Programmer's Reference*, páginas 195–240. Addison Wesley Developers Press, septiembre 1996.

[SC99a] David Sánchez Crespillo, Ricardo Galli, y Yuhua Luo. Un editor 3D interactivo para trabajo cooperativo. En *Actas del Congreso Español de Informática Gráfica*. junio 1999.

[SC99b] Juan Carlos Serra Canals, Ricardo Galli, y Yuhua Luo. Almacenamiento y recuperación remota de escenas 3D. En *Actas del IX Congreso Español de Informática Gráfica*. junio 1999.

[SC00a] David Sánchez Crespillo, Antonio Francisco Bennasar Obrador, Ricardo Galli Granada, y Yuhua Luo. Implementation of mechanisms for concurrent 3D design and visualisation. En *Proceedings of IEEE Symposium on Cooperative Design and Visualization*. julio 2000.

[SC00b] David Sánchez Crespillo, Antonio Bennasar, Juan Fornés, Ricardo Galli, Juan Manuel Huéscar, Juan Carlos Serra, Yuhua Luo, Manuel Gamito, y Marc Albiol. The Multi-Site Editing Tool For Architectural Production. Deliverable 2.1, ESPRIT Project No 26287 M3D, abril 2000.

[SD97] José Miguel Salles Dias, Ricardo Galli, Antonio Carlos Almeida, A. C. Belo, y José Manuel Rebordao. mWorld: A Multiuser 3D Virtual Environment. *IEEE Computer Graphics and Applications*, 17(2), marzo 1997.

[SGI95] Silicon Graphics Inc. The OpenGL Graphics System: A Specification (Version 1.1). Informe técnico, Silicon Graphics Inc, 1995.

[Sil94] A. Silberschatz, J. Peterson, y P. Galvin. *Sistemas operativos. Conceptos fundamentales*. Addison Wesley Iberoamericana, tercera edición, 1994.

[Tem97] Template Graphics Software. *Open Inventor for Win32. IVF. Interactive Visual Framework For Visual C++ and MFC*, 1997.

[Tof] Peter Toft. CVS2HTML. http://www.sslug.dk/cvs2html/.

[VV.94] VV. AA. Computer-Supported Cooperative Work. *Computer*, 27(5), mayo 1994. IEEE Computer Society.

[Wer94a] Josie Wernecke. *The Inventor Mentor*. Addison Wesley Publishing Company, 1994.

[Wer94b] Josie Wernecke. *The Inventor Toolmaker*. Addison Wesley Publishing Company, 1994.