MUNDOS VIRTUALES 3D CON VRML97

Daniel Héctor Stolfi Rosso

Sergio Gálvez Rojas

UNIVERSIDAD DE MÁLAGA

MUNDOS VIRTUALES 3D CON VRML97

Daniel Héctor Stolfi Rosso

Ingeniero Técnico en Informática de Sistemas

Sergio Gálvez Rojas

Doctor Ingeniero en Informática

Dpto. De Lenguajes y Ciencias de la Computación

E.T.S. de Ingeniería Informática

Universidad de Málaga

UNIVERSIDAD
DE MÁLAGA

MUNDOS VIRTUALES 3D CON VRML97

Basado en el proyecto de fin de carrera de Daniel Héctor Stolfi Rosso, tutorizado por el Dr. Sergio Gálvez Rojas

---

AUTORES: Daniel Héctor Stolfi Rosso
Sergio Gálvez Rojas
PORTADA: Daniel Héctor Stolfi Rosso

DEPÓSITO LEGAL: MA-132-2010
ISBN: 978-84-693-0321-4

Índice de contenido

Capítulo 1

Introducción

Este libro pretende introducir al lector en el universo del lenguaje de modelado de realidad virtual (VRML) orientado a Internet, en dónde las páginas web adquieren una tercera dimensión permitiendo que el *internauta* se desplace dentro de los mundos virtuales explorando su contenido mientras se siente inmerso dentro de los mismos.

Ya no es necesario proporcionar varias instantáneas del artículo que desea vender, o describir mediante imágenes animadas cómo funciona determinado ingenio, ni siquiera proporcionar complejos planos que indiquen como encontrar determinado despacho, incluso, hasta se pueden visitar museos sin necesidad de moverse de la silla.

Todo esto y mucho más, nos lo facilitan los mundos diseñados en VRML. Para dominarlos además de las primitivas de diseño se estudiarán técnicas de optimización, dado que se pretende que los modelos puedan ejecutarse en un amplio abanico de computadores, así como técnicas de reducción del tamaño de almacenamiento para potenciar su visualización y uso a través de Internet.

En un principio, se abordará el estudio de las primitivas básicas que permitirán representar cubos, esferas, conos y cilindros. Y luego, se irán introduciendo sentencias para desplazamiento, rotación, escalado, modificación del material y el color de un sólido, iluminación, utilización de elementos multimedia y por último: los elementos dinámicos e interactivos.

También se emplearán *scripts*[1] de código para generar comportamientos complejos y optimizadores escritos de forma específica para ganar fluidez en la ejecución.

Para la construcción de los mundos no son necesarios complejos editores; sólo es necesario contar con un editor de texto plano, y si éste tiene la opción de colorear sintaxis, mejor aún. Además serán imprescindibles un editor gráfico que trabaje con los formatos *.GIF* y *.JPG* y alguno de los *plug-ins*[2] que

[1] Guión o Secuencia de instrucciones que implementa un algoritmo para automatizar algún tipo de proceso. En VRML se pueden usar varios tipos de lenguajes para escribir *scripts* haciendo uso del nodo **Script**. En este proyecto se empleará el lenguaje *vrmlscript* para añadir comportamiento dinámico aunque normalmente los *plug-ins* también soporten *ECMAScript*, también conocido como *Netscape JavaScript.*

[2] Con plug-in se hace referencia a las extensiones que poseen los navegadores web permitiendo incorporar funcionalidades extra generalmente desarrolladas por terceros. Cuando un *plug-in* VRML puede ser utilizado de forma autónoma también se lo denomina *browser VRML*.

se estudiarán en este libro, u otro cualquiera de los existentes siempre y cuando sea capaz de representar VRML en su versión 2.0, más conocido como VRML97.

Como VRML es un lenguaje interpretado no es necesario ningún compilador, de esta manera el fichero de texto que escribamos será el que interprete el navegador a través del *plug-in*.

Si bien, la base teórica nos permitirá intuir qué es lo que estamos viendo cuando nos enfrentamos a unas líneas de código, lo realmente espectacular será contemplar un mundo virtual creado por nosotros mismos comparando el resultado obtenido al visualizarlo en el navegador con lo que teníamos en mente cuando escribíamos el código.

Además con los efectos visuales logrados mediante una iluminación elegida apropiadamente y de los sonidos, unos ambientales y otros de objetos puntuales, se conseguirá una verdadera sensación de realidad que será directamente proporcional al tamaño del monitor y a la correcta disposición de unos altavoces estéreo.

Habiéndonos propuesto el aprendizaje de este lenguaje y antes de tomar contacto con el código en sí mismo, conozcamos un poco más al lenguaje VRML.

Capítulo 2

Estudio del lenguaje VRML

2.1 Introducción

VRML 1 nace en 1995 como un estándar para visitar mundos virtuales a través de Internet. Tiene la virtud de ser un lenguaje que, como HTML (*HyperText Markup Language*[3]) están escritos en ficheros de texto plano, sólo que HTML describe una página web en 2-D, mientras que VRML hace lo propio con un mundo virtual en 3-D.

No es casualidad, entonces, la similitud de ambos nombres; en un principio el significado de VRML se correspondía con *Virtual Reality Markup Language*[4], pero luego fue cambiado al actual: *Virtual Reality Modeling Language*[5] por ser más representativo del carácter de 3-D del lenguaje que de simple texto de etiquetas.

Los mundos virtuales de VRML 1, carecen de vida: No tienen sonido, ni movimiento alguno, sólo permiten visitarlos y desplazarse por ellos. Es por ello que se empiezan a proponer extensiones. La versión 1.1 proponía la inclusión de audio y animaciones, pero no fue llevada a cabo, aunque sí surgieron soluciones propietarias. Por fin en 1997 ve la luz VRML 2.0 también llamado VRML97. Su especificación consta de dos partes:

- ISO/IEC 14772-1 que define funcionalidad base y los tipos de codificación.
- ISO/IEC FDIS 14772-2 que define la funcionalidad y aspectos relacionados con la realización de interfaces externas.

Los objetivos se amplían y ya no sólo es un formato de fichero para describir mundos interactivos por Internet, sino que pretende convertirse en un formato universal para el intercambio de gráficos en 3-D e información multimedia.

Algunos de los criterios de diseño que se han seguido son:

- **Composición:** Que permite el desarrollo por separado de mundos que luego pueden ser combinados o reutilizados. Por ejemplo cada habitante de una ciudad puede hacer su propia casa que posteriormente será referenciada desde un fichero correspondiente a la ciudad que tiene

[3] Lenguaje de Etiquetas de HiperTexto

[4] Lenguaje de Etiquetas de Realidad Virtual

[5] Lenguaje de Modelado de Realidad Virtual

modeladas las calles y las aceras, incluso no es necesario que cada casa se encuentre en el mismo servidor web.

- **Escalabilidad:** Supongamos que en el ejemplo anterior cada casa tiene modelados en su interior los muebles y en la cocina de una se modela una cerilla, a continuación la ciudad se sitúa en el planeta, éste en el sistema solar y éste, a su vez, en la vía láctea. Pues VRML nos permite realizar el modelado a escala real y ver desde cerca la cerilla hasta flotar sobre la galaxia. Los limites sólo son los del Hardware empleado y los de las implementaciones software de los visualizadores.
- **Extensibilidad:** Siguiendo el ejemplo, si alguien construye el modelo de una bicicleta para su casa, todos los que también quisieran una deberían hacer lo mismo. ¿Y si ahora se quisiera otra de un color distinto, o con un manillar diferente?. Para optimizar estas situaciones VRML incorpora los prototipos que no son más que extensiones al lenguaje. Como no existe una primitiva para modelar una bicicleta, alguien la construye y le da opciones de cambio de color y de elección del tipo de manillar, ahora cualquiera que lo desee puede incorporar el prototipo a su modelo, disponiendo el lenguaje ahora de esa primitiva extra. El tiempo ahorrado, se puede emplear para realizar otro prototipo, digamos el de un coche, de esta forma se generarían bibliotecas de prototipos enriqueciendo el lenguaje para los habitantes del mundo del ejemplo.

Debido a la superioridad de VRML97, se hará referencia siempre a primitivas y características de esta versión, sólo cuando se trate de la versión 1.0 se hará la correspondiente aclaración.

Cada fichero VRML consta de un sistema de coordenadas tanto para los objetos definidos en él, como para los que sean incluidos desde una fuente externa. Además deben cumplir con un formato específico, comenzando con la cabecera que se puede observar en el Listado 1.

```
#VRML V2.0 utf8
```

Listado 1: Cabecera VRML97

Ésta tiene carácter de comentario, debido a que empieza con el carácter #, pero también especifica la versión con que estamos trabajando, seguida de la codificación del juego de caracteres: UTF8[6]. Además, el lenguaje es fuertemente tipado. En la Tabla 1 se listan los tipos, tanto para los campos pertenecientes a los nodos como para los eventos.

[6] ISO/IEC 10646-1: Juego de caracteres universal codificado en múltiples octetos, que permite la inclusión de acentos y otros caracteres especiales, incluso japoneses.

Tipo	Descripción
SFBool	Booleano: TRUE o FALSE
SFColor	Tripleta RGB correspondiente al color, cada componente tiene un rango entre 0 y 1.
MFColor	Vector encerrado entre corchetes de 0 o más SFColor separados por comas[7] o espacios.
SFFloat	Número de coma flotante de simple precisión. Se puede utilizar "e" para indicar el exponente.
MFFloat	Vector encerrado entre corchetes de 0 o más MFFloat separados por comas o espacios.
SFImage	Imagen monocroma o en color con transparencia. Véase la Tabla 2.
SFInt32	Entero de 32 bits, si se lo codifica en hexadecimal se lo debe preceder por "0x"
MFInt32	Vector encerrado entre corchetes de 0 o más SFInt32 separados por comas o espacios.
SFRotation	Cuarteto de valores en coma flotante. Los tres primeros indican el eje de rotación normalizado y el cuarto valor es el ángulo de giro en radianes en el sentido que determina la regla de la mano derecha (ver la Figura 1 a continuación)
MFRotation	Vector encerrado entre corchetes de 0 o más SFRotation separados por comas o espacios.
SFString	Cadena de caracteres con formato UTF-8 encerradas entre comillas dobles. Para incluir las comillas dobles dentro de una cadena se debe preceder por la barra invertida.
MFString	Vector encerrado entre corchetes de 0 o más SFString separadas por comas o espacios.
SFTime	Medida de tiempo, en formato de coma flotante, absoluta o relativa. Típicamente especifica el número de segundos desde la fecha origen: 01-01-1970 – 00:00:00 GMT[8].
MFTime	Vector encerrado entre corchetes de 0 o más SFTime separados por comas o espacios.
SFVec2f	Especifica un vector de dos números en formato de coma flotante encerrado entre corchetes.
MFVec2f	Vector encerrado entre corchetes de 0 o más SFVec2f separados por comas o espacios.
SFVect3f	Especifica un vector de tres números en formato de coma flotante encerrado entre corchetes.
MFVect3f	Vector encerrado entre corchetes de 0 o más SFVec3f separados por comas o espacios.

Tabla 1: Tipos de campos y eventos

[7] Por motivos de claridad para la lectura del código fuente se pueden utilizar comas, pero las mismas son interpretadas como si fueran espacios cumpliendo ambos la misma función de separador.

[8] GMT: *Greenwich Mean Time* también conocida como UTC *Universal Time Coordinated* hace referencia a la hora marcada por el Meridiano de Greenwich.

Para determinar el sentido de los ángulos se utiliza, como se menciona en la Tabla 1, la regla de la mano derecha, que consiste en colocar esta mano extendida y con el pulgar apuntando hacia donde lo hace el vector que define el eje de giro y al cerrar los dedos queda determinado el sentido positivo de rotación (véase la Figura 1)

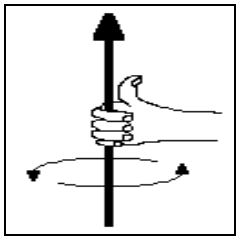


Figura 1: La regla de la mano derecha

Al igual que la cabecera, los comentarios comienzan con el carácter #, y estos pueden empezar en cualquier parte de una línea, extendiéndose hasta el retorno de carro.

Los nodos que conforman un fichero *.WRL*[9], se estructuran como un árbol, donde queda reflejado el carácter jerárquico del lenguaje, luego, al reutilizar las primitivas el árbol se convertirá en un grafo. Por ejemplo, para representar una caja y un cono desplazados del origen, el esquema sería como el que se aprecia en la Figura 2.

El círculo representa a un nodo de agrupación, los rectángulos de vértices curvos a los nodos desplazamiento, el cuadrado a la primitiva de la caja y el triángulo a la del cono. Nótese que esta convención se toma únicamente para este libro.

Un aspecto muy importante también sobre el lenguaje es que es sensible a mayúsculas y minúsculas, es decir que **Box** es correcto como nombre de primitiva mientras que *box* producirá un error, si se lo emplea para ese mismo fin.

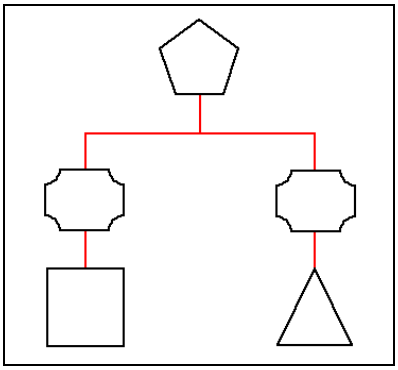


Figura 2: Árbol representando un fichero .WRL

[9] WRL: Abreviatura de *world* (mundo), es la extensión empleada para los ficheros VRML sin compresión.

2.2 Diferencias entre versiones

Como se adelantaba más arriba, VRML 1.0 definía mundos estáticos, sin contenido multimedia. VRML97 proporciona primitivas para añadir sonido, vídeo y elementos interactivos.

Un intento de ampliación de la versión 1.0 fue acometido por **Netscape Communications Corporation** para su navegador VRML: Live3D[10]. Estas extensiones, que llevaban el nombre del navegador VRML, permitían entre otras cosas agregar imágenes o colores de fondo, utilizar texturas animadas, definir el estilo de las colisiones, inclusión de sonidos, etc. Todas estas mejoras forman parte ya de VRML97 y están disponibles en la mayoría de los *plug-ins* que siguen el estándar.

En cuanto al código fuente, un fichero que defina a un cubo de color verde en VRML 1.0 sería de esta manera:

```
#VRML V1.0 ascii

Separator {
      Material {
            diffuseColor      0 0.8 0
      }
      Cube {
            width 20
            height      1
            depth 20
      }
}
```

Listado 2: Cubo Verde en VRML 1.0

Mientras que en VRML97 quedaría así:

```
#VRML V2.0 utf8

Shape {
   appearance Appearance {
      material Material {
         diffuseColor   0 0.8 0
      }
   }
   geometry Box {
      size     20 1 20
   }
}
```

Listado 3: Cubo verde en VRML97

Si bien se mantiene la estructura jerárquica, VRML97 apuesta por una sintaxis que podría denominarse, haciendo una similitud con la programación orientada a objetos, de "constructores"; por ejemplo el campo ***appearance*** del nodo **Shape** va seguido de un constructor del mismo nombre pero con la primera letra mayúscula: **Appearance** { } y dentro de su ámbito, delimitado por las llaves, tiene sus propios campos, entre ellos, el que aparece en el ejemplo: ***material*** en el cual se repite el esquema: campo Constructor { campos }.

[10] El desarrollo de este navegador se ha estancado en el año 1997 al salir la nueva versión de VRML.

Tanto **Appearance** como **Material** poseen más campos que los que se utilizan en el ejemplo, pero los que se omiten toman sus valores por defecto. Para más detalles véase la sintaxis completa en los Puntos 3.3.1 y 3.3.1.1 respectivamente.

A pesar de la ampliación de funcionalidades que VRML97, los defensores de la versión 1.0 sostienen que esta versión tiene una sintaxis un poco más sencilla, que existen algunos *plug-ins* para los navegadores que no soportan VRML97 y sí la versión anterior, y que los que la soportan también poseen conversores de VRML97 a VRML 1.0 para permitir visualizar los modelos hechos en la versión antigua.

2.3 Herramientas de visualización

De los navegadores VRML que existen en la actualidad destacan cuatro:

2.3.1 Cortona VRML Client (http://www.parallelgraphics.com)

En su versión actual, 5.1, Cortona VRML Client es un *plug-in* para Internet Explorer y Netscape Navigator bajo plataformas win32.

Como características principales permite usarse tanto con Direct3D como con OpenGL, aunque con estas bibliotecas presenta una calidad de imagen bastante pobre. El comportamiento es bueno en general, siendo la mejor alternativa en la actualidad. No tiene extensiones propias y presenta ciertos problemas de fluidez frente a un modelo muy grande. Se observa también un error en la precisión en cuanto al cálculo de distancias para sólidos alejados del observador cuando se encuentran muy cerca uno de los otros, produciéndose a veces problemas de solapamiento y transparencias no deseadas.

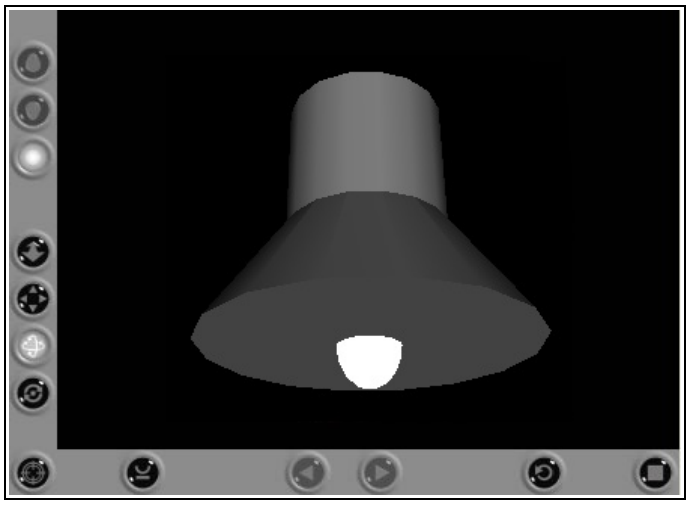


Figura 3: Cortona

Entre sus virtudes se destacan una opción para no presentar el mundo virtual hasta que no se haya cargado en su totalidad y la posibilidad de modificar el comportamiento a través del mismo código HTML de la página desde donde se carga al *plug-in*. Además, con objeto de mejorar la fluidez de los

movimientos, permite un degradado de la calidad de la presentación mientras exista un movimiento en pantalla, para luego aplicar filtros de calidad a las texturas cuando el visitante permanezca estático.

Otras aplicaciones de Parallelgraphics relacionadas con VRML son:

- **VRML 1.0 to VRML97:** Traductor de ficheros.
- **VrmlPad:** Editor profesional para ficheros VRML.
- **Extrusion Editor:** *Plug-in* para VrmlPad para crear nodos **Extrusion**.
- **Internet Model Optimizer:** Optimiza modelos complejos en 3-D creados desde sistemas *CAD/CAM/CAE*[11] para su uso en Internet.
- **Internet Space Builder:** Editor que crea fácilmente mundos en 3-D.

2.3.2 Blaxxun Contact 3-D[12] (http://www.blaxxun.com/)

Este *plug-in* además de servir como navegador VRML, provee una variedad de extensiones propietarias que permiten la colaboración en línea, navegación conjunta, construcción de comunidades virtuales, chat[13] con sonidos personalizados y *avatar*[14] visible con aspecto y gesticulaciones configurables.

Está disponible tanto para Internet Explorer como para Netscape Communicator[15] en plataforma Windows y es capaz de interpretar ambas versiones de VRML, actualmente se encuentra en la versión 5.104.

Trabaja con las bibliotecas OpenGL y Direct3D, tiene una calidad de imagen aceptable, permitiendo alisar texturas, visualizar en marcos de alambre (*wireframe*) y otras optimizaciones. Por contra, los colores no son tan vivos como en otros *plug-ins*.

[11] CAD: *Computer-Aided Design* – Diseño Asistido por Computador. CAM: *Computer-Aided Manufacturing* – Fabricación Asistida por Computador. CAE: *Computer-Aided Engenering* – Ingeniería Asistida por Computador.

[12] Blaxxun es una marca registrada de Blaxxun Technologies.

[13] Del inglés charla, se emplea para hacer alusión a la comunicación por Internet entre dos o más personas, que normalmente intercambian mensajes de texto, realizando una conversación escrita.

[14] El avatar es la representación abstracta del usuario del mundo virtual.

[15] Netscape Comunicator es una marca registrada de Netscape Communications Corp.

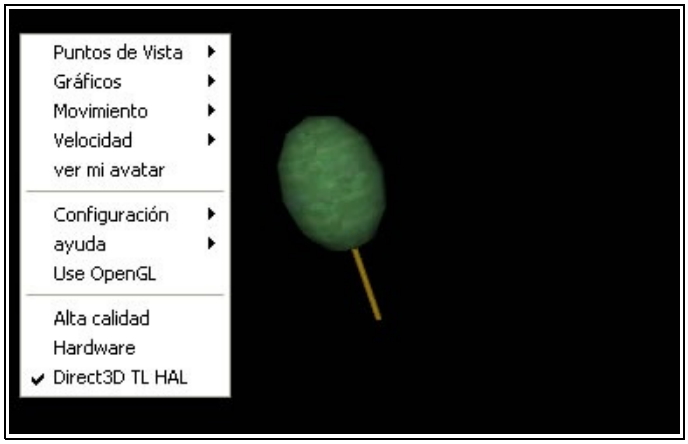


Figura 4: Blaxxun Contact 3-D

2.3.3 Octaga Player (http://www.octaga.com/)

Se trata de un navegador VRML autónomo, es decir que no se ejecuta embebido dentro de un navegador web. No tiene el nivel de funcionalidad de los dos anteriores, pero permite visualizar la mayoría de los escenarios con los que se ha probado. Su interfaz de manejo no es del todo clara, llegando a ser incluso poco intuitiva.

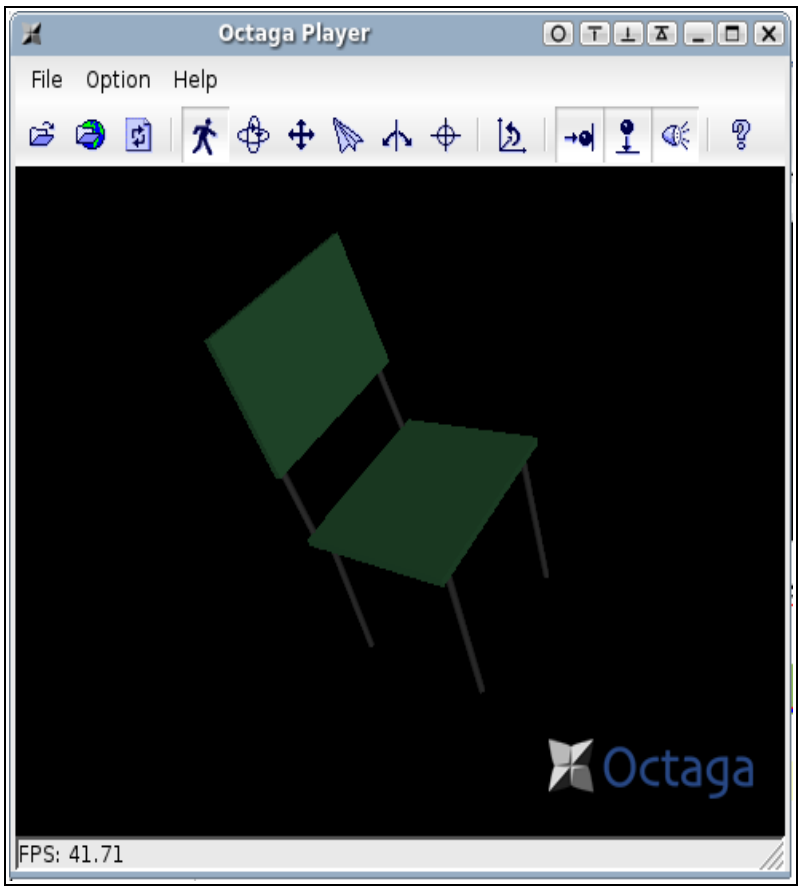


Figura 5: Octaga

Como curiosidad destacar que se encuentra disponible tanto para Linux como para win32.

2.3.4 Freewrl (http://freewrl.sourceforge.net/)

La opción *OpenSource*[16] que se encuentra en su versión 1.19.6, se integra tanto como programa autónomo que se puede ejecutar desde la línea de comando como *plug-in* para los navegadores Mozilla-Firefox, Konqueror, etc.

Dependiendo de la distribución de linux que se posea (existen paquetes para Fedora, Ubuntu, Debian, Suse y Gentoo) dará más o menos trabajo hacer funcionar la versión precompilada, o bien se deberá compilar el código fuente. Una vez conseguido esto, y dados de alta los *plug-ins* en el navegador, al abrir un fichero *.WRL* o visitar una página con contenido VRML se visualizará el contenido en pantalla permitiendo la navegación por el mundo virtual.

El rendimiento es bastante malo, sobre todo cuando no se trata de escenarios sencillos, incluso llegando a no comenzar nunca la visualización. Pero por otro lado, hay que tener en cuenta que está desarrollado por particulares en su tiempo libre (el proyecto lo comenzó Tuomas J. Lukka, y ahora lo continúan John Stewart y otros) y que además permite la integración con gafas de visión en 3-D y otras interfaces de detección de movimiento.

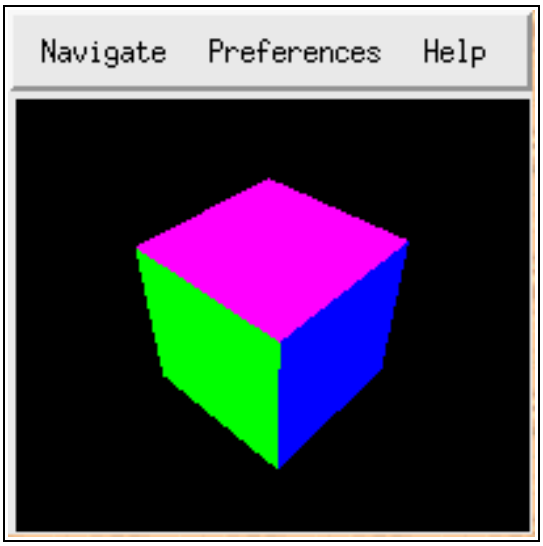


Figura 6: Freewrl

Otro desarrollo *OpenSource* es **OpenVRML** (http://www.openvrml.org/) que además del *plug-in* ofrece bibliotecas disponibles para ser utilizadas desde aplicaciones escritas por el usuario.

[16] Licencia de código abierto que no sólo impide la venta del software que se acoge a la misma, si no que también especifica la libre distribución del código fuente, promoviendo la no discriminación de personas o grupos ni en el ámbito de uso, sea tanto personal, empresarial o en investigación. (http://www.opensource.org)

2.4 Introducción a las primitivas más importantes

Es conveniente en este punto empezar a familiarizarnos con el lenguaje en sí, para ello empezaremos a ver código y los resultados que con éste se obtienen. Como primer paso se presentarán las primitivas de dibujo básicas de VRML, que se utilizan para modelar un paralelepípedo o caja (**Box**), una esfera (**Sphere**), un cono (**Cone**) y un cilindro (**Cylinder**) respectivamente.

Para crear una caja se necesita definir el tamaño (***size***) en sus tres dimensiones: anchura, altura y profundidad. Por ejemplo:

```
Box {
   size  2 0.5 1
}
```

Listado 4: Primitiva Box

Esta porción de código define una caja de dos metros de ancho, medio metro de alto y un metro de profundidad.

Para dibujar primitivas, en VRML, es necesario un "contenedor", éste es para todos los casos el nodo **Shape**; en el cual, además de la primitiva a dibujar, se debe especificar su color o textura. De esta manera **Shape** define tanto la geometría (***geometry***) como la apariencia (***appearance***) de la primitiva a modelar.

El código completo para crear una caja blanca de 2 x 0,5 x 1 metros sería el presentado en el Listado 5, obteniendo el resultado que se observa en la Figura 7. Obsérvese también los efectos de la iluminación sobre la caja produciendo el sombreado de sus caras laterales.

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {
            material Material {
                  diffuseColor 1 1 1
            }
      }
      geometry Box {
            size 2 0.5 1
      }
}
```

Listado 5: Caja blanca de 2 x 0.5 x 1

El constructor para el campo ***appearance*** se analizará más adelante, en el Punto 3.3.1, de momento sólo será necesario destacar que el campo ***diffuseColor*** requiere como parámetro las tres componentes del color, conocidas como **RGB**[17] que describen el porcentaje de rojo, verde y azul que contiene el color, en un rango entre 0 (0%) a 1 (100%).

[17] Correspondiéndose con la primera letra de *Red* (rojo), *Green* (verde) y *Blue* (azul).

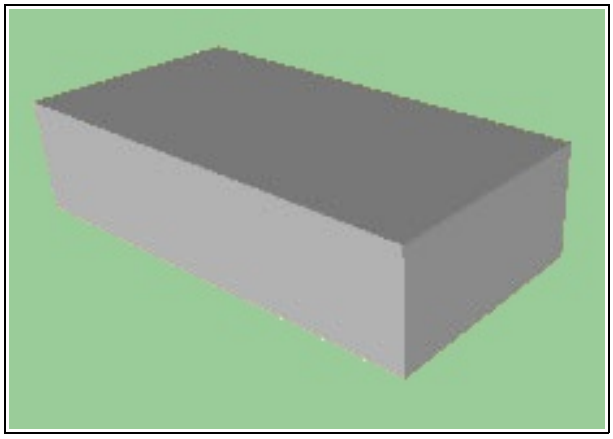


Figura 7: Caja Blanca de 2 x 0.5 x 1

Esta representación para los colores, obedece a la forma en que se representan los mismos en la pantalla del monitor. Éste consta de tres cañones que componen los colores emitiendo un haz de electrones, que según la intensidad con la que choca contra la superficie de fósforo depositada en la cara interior de la pantalla[18] correspondiente a cada uno de los tres colores, se va a producir una mezcla cromática, produciendo el color deseado.

Por ejemplo un color RGB igual a 1 1 1, es decir las tres componentes al 100% se corresponderá con el color blanco, un RGB de 0 0 0, es decir las tres al 0% será el color negro. Si sólo se tiene componente R (1 0 0) serán tonalidades de rojo desde el más oscuro al más claro, a medida que se va pasando desde 0 a 1. Lo mismo para la componente G y B dando tonalidades verdes y azules respectivamente. Al mezclarse se obtendrán más colores, por ejemplo un RGB igual a 1 1 0, será amarillo, 0 1 1, será cian y 1 0 1 magenta.

Haciendo referencia ahora al campo ***geometry***, del nodo **Shape**, éste debe ir siempre seguido por la primitiva que se va a modelar o por una referencia a la misma.

Además siempre será necesario incluir la cabecera del fichero *.WRL* (ver Listado 1) para que el *plug-in* interprete correctamente el contenido de éste. En algunos de los ejemplos de este proyecto, cuando se trate de trozos de código, no se la incluirá por tratarse de una sección de un fichero más grande del que se quiere resaltar alguna particularidad.

No ocurre lo mismo con los campos de los nodos, ya que si se omiten, toman sus valores por defecto. De esta manera una caja definida como se observa en el Listado 6 se corresponderá con una caja que medirá un metro de ancho, un metro de alto y un metro de profundidad.

[18] Se hace la descripción basándose en el funcionamiento de un monitor con pantalla de tubo de rayos catódicos (CRT - *Cathodic Ray Tube*), para el caso de una pantalla de cristal líquido (LCD – *Liquid Cristal Display*) la separación del color en componentes es la misma, pero el elemento emisor de luz no es una capa de fósforo, sino un transistor de película delgada (TFT – *Thin Film Transistor*).

```
Box { }
```

Listado 6: Caja de tamaño por defecto

La ubicación en el espacio la proporcionará el nodo **Transform** como se verá luego, por el momento sólo cabe decir que la ubicación por defecto es en el centro del sistema de coordenadas[19], X = 0, Y = 0, Z = 0, como queda representado a continuación, en la Figura 8.

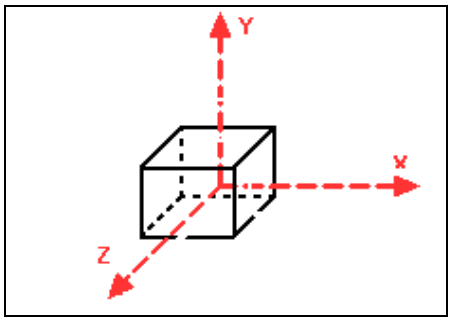


Figura 8: Caja centrada en el origen de coordenadas

El esquema se repite para el caso de la esfera, sólo que en este caso el campo a especificar es el radio (***radius***); por supuesto también se puede prescindir de él como en el caso de la caja, y en este caso lo que se obtendría sería una esfera de un metro de diámetro.

El código para representar una esfera de radio igual a medio metro y de color rojo es el que se aprecia en el Listado 7.

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {
      material Material {
                  diffuseColor 1 0 0
            }
      }
      geometry Sphere {
            radius 0.5
      }
}
```

Listado 7: Esfera roja de radio 0.5

Como se aprecia levemente en la Figura 9, la esfera está formada por polígonos, aunque normalmente pasan desapercibidos, en algunos casos, especialmente en esferas de gran tamaño producen un efecto un poco desagradable. VRML no contempla la posibilidad de definir cuantos de estos polígonos se utilizan, o dicho de otro modo, la “suavidad” con que se modela la esfera.

[19] VRML utiliza un sistema de coordenadas tridimensional en el que se corresponden el eje X con el ancho, el eje Y con la altura y el eje Z con la profundidad; tomándose como referencia el punto de vista inicial por defecto.

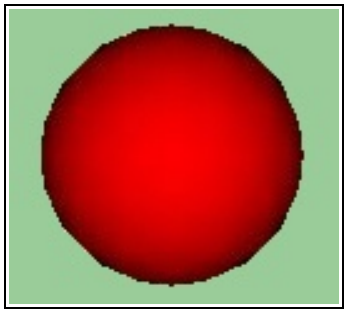


Figura 9: Esfera roja de radio 0.5

También en este caso, la esfera se encuentra centrada en el origen de coordenadas porque no se ha especificado una traslación, ver Figura 10.

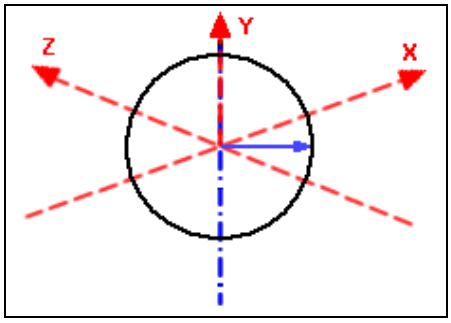


Figura 10: Esfera centrada en el origen de coordenadas

Para dibujar un cono (**Cone**) se utiliza su primitiva, especificando como campos el radio de su base (***bottomRadius***), su altura (***height***) y dos campos booleanos: ***side*** que indica si se dibuja la generatriz del cono y ***bottom*** que se refiere a si se dibujará su base, ver Listado 8.

En este punto hay que tener en cuenta que estas primitiva junto con el cilindro, aún por estudiar, sólo realizan el renderizado de sus caras exteriores, o sea que si miramos dentro de un cono al que le falta la base (***bottom*** **FALSE**) no veremos nada, porque la generatriz sólo será visible en su cara exterior. Este comportamiento se puede modificar cuando dibujamos caras aisladas utilizando los nodos **IndexedFaceSet** o **Extrusion** (véanse los Puntos 3.2.6 y 3.2.10 respectivamente) que sí permiten elegir el lado que se renderiza o incluso si son o no visibles ambos lados de la figura.

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {
      material Material {
         diffuseColor 0 1 0
      }
   }
   geometry Cone {
      bottomRadius  0.5
      height      1
   }
}
```

Listado 8: Cono verde de 1 metro de altura

En este caso la representación será de un cono verde, de un metro de altura y medio metro de radio en su base, ver Figura 11.

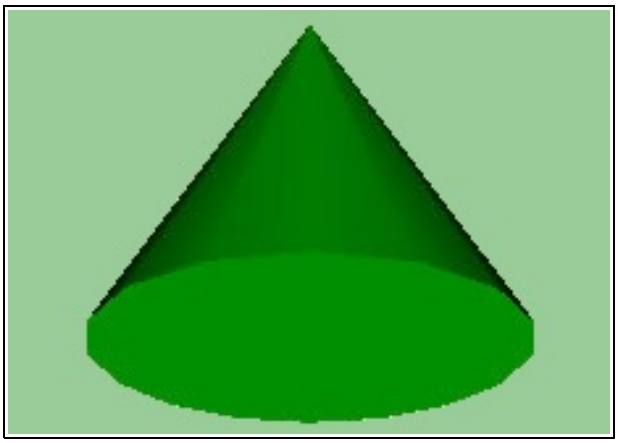


Figura 11: Cono verde de 1 metro de altura

En el ejemplo no se han especificado los valores de ***bottom*** y de ***side*** porque por defecto valen **TRUE**, es decir, que si no se especifica lo contrario, se dibujará el cono completo. El radio de la base por defecto es de un metro y la altura de 2.

Como se ve en la Figura 12, el cono se dibujará centrado en (0,0,0) igual que las figuras de los ejemplos anteriores, tomando como punto central el que se encuentra a la mitad de la altura del eje proyectado desde el centro de la base.

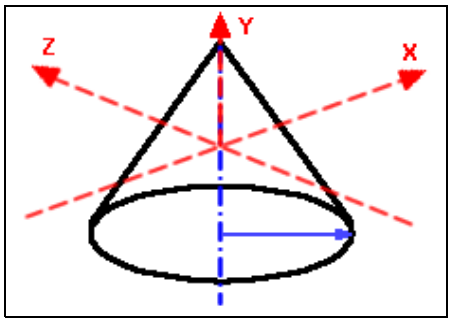


Figura 12: Cono centrado en el origen de coordenadas

La primitiva que resta describir es la correspondiente al cilindro (**Cylinder**); la misma es similar al cono ya que consta también de un radio (***radius***) que es común a la cara circular superior e inferior, la altura de la generatriz (***height***), el booleano que decide si se la dibuja o no (***side***), y otros dos booleanos para hacer lo propio con la cara circular inferior (***bottom***) y la superior (***top***).

El Listado 9 representa un cilindro azul de un metro de altura y de una base de medio metro de radio, con la particularidad de que no se dibujará la parte superior. Y es que, por defecto, se dibujan todas las superficies del cilindro, salvo que se especifique lo contrario, como es el caso del ejemplo (véase en la Figura 13). El valor por defecto del radio es de un metro y el de la altura es de 2.

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {
      material Material {
         diffuseColor 0 0 1
      }
   }
   geometry Cylinder {
      radius   0.5
      height   1
      top   FALSE
   }
}
```

Listado 9: Cilindro Azul de un metro de altura sin cara superior

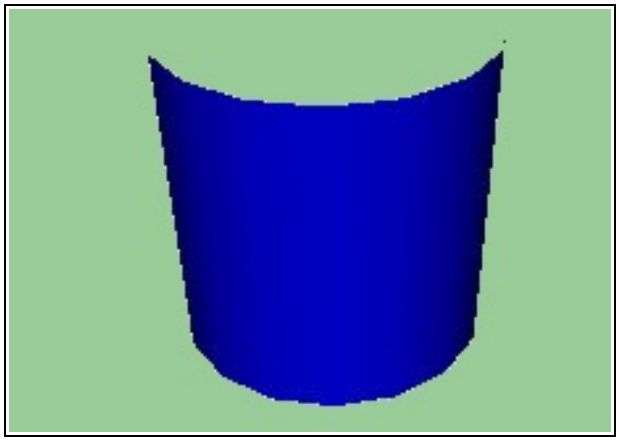


Figura 13: Cilindro azul sin cara superior

También en este caso se dibujará centrado en el origen de coordenadas, como se aprecia en la Figura 14, tomando como criterio para calcular el punto central del cilindro el mismo que se ha empleado para el cono.

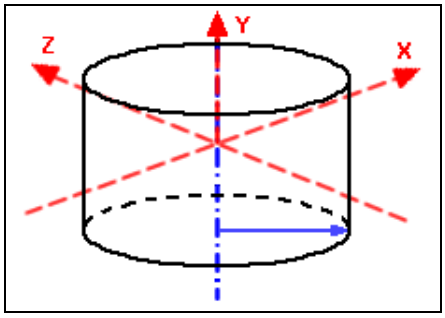


Figura 14: Cilindro centrado en el origen de coordenadas

Con estas cuatro primitivas, ya se tiene una breve noción de cómo se construye un mundo virtual en VRML; en un principio puede parecer que con cajas, esferas, conos y cilindros no se puede modelar objetos de la vida cotidiana, sin embargo, para cuando sea necesario, existen otros nodos que permiten formas más complejas como **IndexedFaceSet** y **Extrusion** por ejemplo. Con ellos se pueden construir sólidos por medio de caras, o bien en base a una sección transversal y una curva sobre la cual ir dibujándola formando así un sólido de revolución u otro tipo de superficies.

Aquí ya comienzan a presentarse algunas de las carencias de VRML97, tanto la esfera, como el cono y el cilindro poseen sus zonas "curvas" hechas por medio de polígonos reduciendo la precisión en la representación de cada una de las formas.

Nótese que el color de la superficie de las primitivas no es uniforme, sino que obedecen a la distribución de la luz emitida desde el *avatar* la cual hace las veces de un casco de minero. Como estos mundos son muy sencillos no poseen luz propia (ni fondo) por lo que es necesario tener encendida esta luz, llamada en VRML *headlight*[20] de lo contrario no se visualizaría figura alguna[21].

Por motivos de claridad en las capturas se ha usado fondo claro, pero en general, el fondo por defecto es de color negro.

2.5 Resumen

En este capítulo, hemos aprendido un poco sobre la historia de VRML, los motivos de su nacimiento y sus tipos de datos. Luego, se ha hecho una comparativa donde se puede apreciar la evolución que significa la versión 2.0, llamada VRML97, frente a su precursora y, hoy día, obsoleta versión 1.0.

Posteriormente se han presentado algunos de los *plug-ins* disponibles en el mercado y se ha efectuado una breve valoración de cada uno y donde hoy por hoy queda claro que frente a una comparativa, el ganador es Cortona VRML Client (http://www.parallelgraphics.com) por calidad y eficiencia en la representación aunque, si bien es cierto, no posee extensiones propias está en continuo desarrollo.

Por último, hemos dado los primeros pasos con VRML abordando algunas de las posibilidades que posee en cuanto a la representación de primitivas. Y hemos visto superficialmente la elección de los materiales y los efectos que producen las luces sobre los sólidos en base al material con el que están modelados.

[20] Headlight: Luz que proviene desde la cabeza del avatar. Se la conoce como la luz del casco del minero.

[21] Existe un campo del nodo **Material** llamado **emissiveColor** que permite, como se verá más adelante definir el color que "emite" una primitiva. De esta manera, en ausencia de fuente de luz (del modelo y la *headlight*) sólo se podrán ver las que tengan definido este color que por defecto, y en los ejemplos expuestos hasta ahora, es negro (RGB = 0 0 0).

Capítulo 3

Manual descriptivo de VRML

3.1 Introducción

En este capítulo se abordará la descripción de la totalidad de los nodos que comprenden al lenguaje. Para cada uno se incluirá información sobre su sintaxis, una explicación detallada de cada campo y uno o varios ejemplos de uso[22].

Por cuestiones pedagógicas, los nodos se han dispuesto en grupos comunes intentando facilitar su entendimiento, evitando ser sólo una guía de consulta, aspirando al carácter de tutorial.

En la descripción sintáctica, junto con cada miembro de un nodo, se especificará si se trata de un evento o de un campo, el tipo de evento o campo (véase en la Tabla 1 los tipos de VRML y su descripción), el valor por defecto que toma y el rango de valores que se le pueden asignar.

Los campos de tipo evento (*eventIn* y *eventOut*) que pertenecen a cada nodo serán explicados en conjunto en el Punto 3.7 junto con los sensores, debido a que favorecerá a su comprensión por estar íntimamente ligados.

3.2 Primitivas

Con este nombre se va a englobar a todos los nodos que producen una sólido o una figura. Algunas de ellas ya se han introducido en el Punto 2.4, pero aquí se expondrán con más detalle y abordando todos sus aspectos y variantes sintácticas.

3.2.1 Shape

Sintaxis:

```
Shape {
      exposedField SFNode appearance NULL
      exposedField SFNode geometry   NULL
}
```

El nodo **Shape** es un contenedor que sirve para definir tanto la apariencia (***appearance***) de una primitiva, como la geometría (***geometry***) del sólido o la figura a dibujar. Por sí solo no produce ninguna

[22] Para una mayor claridad en las figuras se dispondrá del fondo del color más apropiado, para facilitar la correcta visualización del modelo analizado. No obstante, téngase en cuenta que por defecto en VRML el fondo es de color negro.

salida ya que los valores por defecto de sus dos campos son nulos. Pero por otro lado es el paso previo en la jerarquía para poder modelar cualquier sólido o figura dentro del mundo.

El campo ***appearance*** requiere a continuación un constructor de apariencia que corre a cargo del nodo **Appearance**, el cual se analiza en el Punto 3.3.1 y cuya función será especificar el color y otras características de la primitiva que se va a modelar. No confundir el campo ***appearance*** del nodo **Shape** con el nodo **Appearance**. Esta dualidad diferenciada por la mayúscula se da en varias ocasiones más en la sintaxis de VRML.

Al campo ***geometry*** le seguirá un nodo y sólo uno de los que se estudiarán a continuación, definiendo así qué primitiva será la modelada. Si no se especifica este campo, el nodo **Shape** no producirá ningún efecto sobre el mundo que se está creando.

3.2.2 Box

Sintaxis:

```
Box {
      field    SFVec3f size  2 2 2        # (0,∞)
}
```

Representa en el mundo virtual una caja o paralelepípedo, de dimensiones especificadas por su campo ***size*** (tamaño). Por defecto toma el valor de 2 x 2 x 2 metros, pudiendo ser cada componente cualquier valor mayor que 0, tal cual se puede observar en la definición de rango dentro de la sintaxis.

Al modelarse centrada en el origen de coordenadas local[23], la caja distribuirá el valor de sus componentes a ambos lados de los ejes, es decir, para el caso por defecto, la intersección con los ejes de coordenadas se dará para x=1, x=-1; y=1 , y=-1; z=1, z=-1, véase la Figura 15.

Según el estándar, los sólidos sólo se renderizarán para ser vistos desde fuera, si se observa un nodo box desde dentro, los efectos serán impredecibles. Esta es una de las libertades (o lagunas) que deja la especificación de VRML, para que sean los programadores de los *plug-ins* los encargados de decidir el comportamiento en estos casos.

[23] Se hace referencia al origen de coordenadas local, haciendo la distinción del origen de coordenadas del mundo virtual, debido a que mediante las transformaciones que se verán junto con el nodo **Transform** se puede desplazar y/o rotar a un conjunto de nodos variando de esta manera el origen de coordenadas local a los mismos de forma relativa al origen de coordenadas del mundo virtual, el cual siempre permanece inalterable.

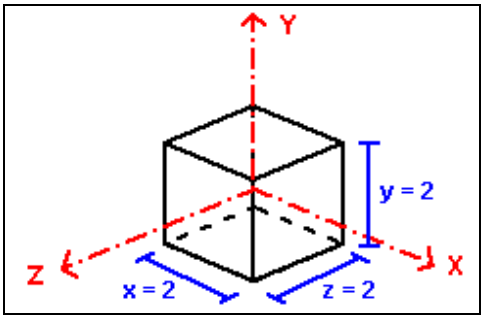


Figura 15: El nodo Box

En el Listado 10 se observa un ejemplo de empleo del nodo **Box** dentro del código fuente modelado con sus valores por defecto y en la Figura 17 el resultado obtenido. Nótese que se ha utilizado para el campo ***appearance*** del nodo **Shape** su valor por omisión, lo que producirá un cubo de material blanco sin sombreado alguno (véase en el Punto 3.3.1 para consultar los valores predeterminados del constructor **Appearance)**

```
#VRML V2.0 utf8
Shape {
   geometry Box { }
}
```

Listado 10: El nodo Box

3.2.3 Sphere

Sintaxis:

```
Sphere {
      field SFFloat radius  1    # (0, ∞)
}
```

Este nodo se utiliza para modelar una esfera centrada en el origen de coordenadas. Su campo ***radius*** (radio) toma el valor por omisión de 1 metro, debiendo siempre ser mayor que 0. Nuevamente, este nodo sólo renderizará la cara exterior de la esfera, siendo impredecible el comportamiento cuando se mire desde dentro de ella. Véase en la Figura 16 la representación del nodo **Sphere** centrado en el origen de coordenadas local.

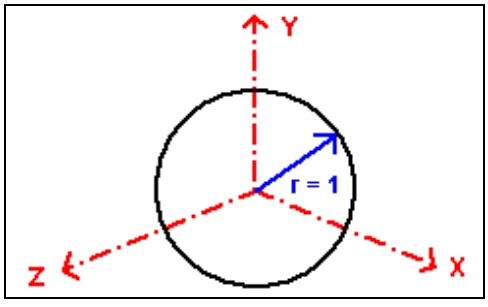


Figura 16: El nodo Sphere

En el Listado 11 se emplea el nodo **Sphere** en el código fuente que modela una esfera con radio por defecto cuyo resultado se aprecia en la Figura 17, para el campo ***appearance*** del nodo **Shape** también se empleó su valor predeterminado (véase en el Punto 3.3.1, los valores por defecto del constructor **Appearance).**

```
#VRML V2.0 utf8
Shape {
   geometry Sphere { }
}
```

Listado 11: El nodo Sphere

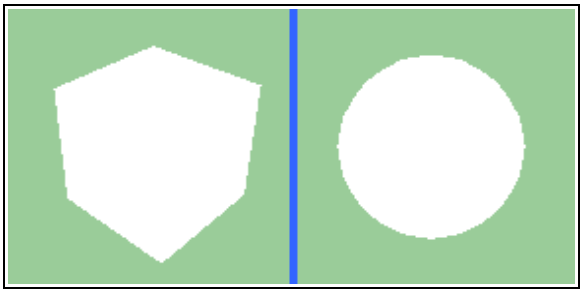


Figura 17: Los nodos Box y Sphere

3.2.4 Cylinder

Sintaxis:

```
Cylinder {
      field    SFBool    bottom  TRUE
      field    SFFloat   height  2         # (0, ∞)
      field    SFFloat   radius  1         # (0, ∞)
      field    SFBool    side    TRUE
      field    SFBool    top     TRUE
}
```

El nodo **Cylinder** permite construir un cilindro centrado en el origen de coordenadas local, tomando como valores por defecto para sus campos ***height*** (altura) y ***radius*** (radio) los valores de 2 metros y 1 metro respectivamente. Estos campos deben consistir siempre en valores mayores que 0.

Existen también tres campos del tipo booleano que indicarán el renderizado o no de las distintas caras del cilindro. De esta manera, el campo ***bottom*** hará referencia a la cara inferior del cilindro, el campo ***top*** se referirá a la superior y el campo ***side*** hará lo propio con la generatriz. Un valor igual a TRUE redundará en que se rendericen estas caras y un valor igual a FALSE producirá un cilindro al cual le faltarán las caras correspondientes a los campos que tomen este valor. En la Figura 18 se presenta el esquema de un cilindro centrado en el origen de coordenadas local. Si se construye el cilindro sin la cara superior o inferior, el *avatar* no colisionará con la misma, pudiendo introducirse dentro de esta primitiva, siempre y cuando las dimensiones lo permitan. En esta situación no podrán verse las caras interiores debido a que no son renderizadas por la mayoría de los *plug-ins*.

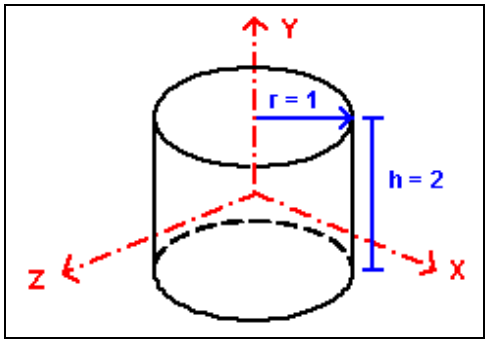


Figura 18: El nodo Cylinder

En el Listado 12 se observa el código fuente empleando el nodo **Cylinder** modelado con el aspecto y tamaño predeterminados y en la el resultado que se obtiene al visualizarse en el *plug-in*.

En el Punto 3.3.1 se pueden consultar los valores por defecto del constructor **Appearance**.

```
#VRML V2.0 utf8
Shape {
   geometry Cylinder { }
}
```

Listado 12: El nodo Cylinder

3.2.5 Cone

Sintaxis:

```
Cone {
      field     SFFloat   bottomRadius 1        # (0, ∞)
      field     SFFloat   height       2        # (0, ∞)
      field     SFBool    side         TRUE
      field     SFBool    bottom       TRUE
}
```

El nodo **Cone** se utiliza para modelar un cono en el mundo virtual. Por defecto el campo ***height*** toma el valor de 2 metros y el campo ***bottomRadius*** será de 1 metro, representando un cono de 2 metros de altura y con una base circular de 1 metro de radio, centrado en el origen de coordenadas, véase la Figura 19. Tanto la altura como el radio deben ser siempre valores mayores que 0.

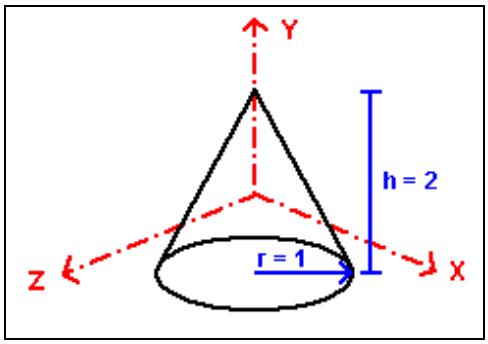


Figura 19: El nodo Cone

Mediante los campos ***side*** y ***bottom*** del tipo booleano se indicará si se renderizará o no la generatriz o la cara inferior del cono, dependiendo de que tomen el valor TRUE o FALSE respectivamente.

Las caras del cono se renderizarán sólo en su lado exterior, si se ven desde dentro los efectos obtenidos no están definidos por la especificación.

Como ejemplo del uso del nodo **Cone** de dimensiones predeterminadas se tiene al Listado 13 donde también se emplea el aspecto por defecto para el campo ***appearance*** del nodo **Shape**. Véase en la Figura 20 el resultado producido por este listado y en el Punto 3.3.1 los valores por omisión del constructor **Appearance**.

```
#VRML V2.0 utf8
Shape {
   geometry Cone { }
}
```

Listado 13: El nodo Cone

En la Figura 20 se observan a los nodos **Cylinder** y **Cone** modelados con el material por defecto. Nótese que las características de este material provocan que cada nodo ignore a las fuentes de luz no produciendo reflejos, por lo que el efecto visual obtenido no permite diferenciar sus aristas al carecer de sombras. Se ha optado por esta solución para no eclipsar el elemento de estudio actual, las primitivas, incluyendo información correspondiente al aspecto visual.

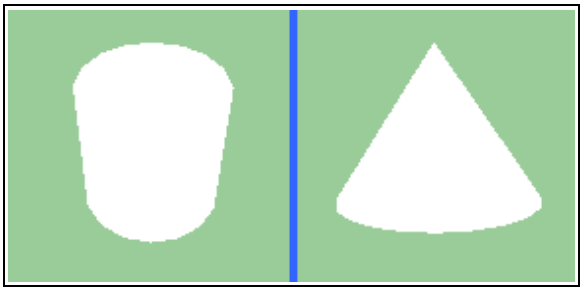


Figura 20: Los nodos Cylinder y Cone

3.2.6 IndexedFaceSet

Sintaxis:

```
IndexedFaceSet {
      eventIn       MFInt32 set_colorIndex
      eventIn       MFInt32 set_coordIndex
      eventIn       MFInt32 set_normalIndex
      eventIn       MFInt32 set_texCoordIndex
      exposedField  SFNode  color             NULL
      exposedField  SFNode  coord             NULL
      exposedField  SFNode  normal            NULL
      exposedField  SFNode  texCoord          NULL
      field         SFBool  ccw               TRUE
      field         MFInt32 colorIndex        []        # [-1, ∞)
      field         SFBool  colorPerVertex    TRUE
      field         SFBool  convex            TRUE
      field         MFInt32 coordIndex        []        # [-1, ∞)
      field         SFFloat creaseAngle       0         # [0, ∞)
```

```
      field         MFInt32 normalIndex       []        # [-1, ∞)
      field         SFBool  normalPerVertex   TRUE
      field         SFBool  solid             TRUE
      field         MFInt32 texCoordIndex     []        # [-1, ∞)
}
```

Este nodo, define una cara sólida en el espacio. Para ello utiliza una lista de vértices definida en el campo ***coord*** por medio del constructor **Coordinate (**véase el Punto 3.2.6.1). Estos vértices serán indexados por el campo ***coordIndex*** definiendo el orden en el que se utilizarán para la construcción de la cara. Se puede definir más de una cara por nodo utilizando sublistas de coordenadas, teniendo en cuenta que para terminar cada cara se colocará como índice el valor –1 para continuar con la enumeración de vértices para la cara siguiente.

Al primer punto de la lista de coordenadas le corresponde el índice número 0, al segundo el 1, etc. De esta manera el número mayor para el campo ***coordIndex*** no debe superar al número total de coordenadas menos uno.

Por defecto la cara se renderizará de un solo lado, el que se define siguiendo el orden de los vértices en sentido contrario al de las agujas del reloj, el otro lado no será representado pudiéndose ver a través del mismo como si no hubiese ninguna entidad en ese sitio. Para las colisiones sí se lo tomará en cuenta impidiendo atravesar la cara definida por el nodo aunque no sea visible desde ese lado.

Para modificar esta característica existen dos campos: uno es el encargado de definir cual cara se renderiza y se trata del campo ***ccw*** que es del tipo booleano, y si toma el valor TRUE significa que se renderizará la cara definida por sus puntos en el orden del sentido contrario al de las agujas del reloj y si vale FALSE lo será la cara del otro lado. El otro campo en cuestión dejará sin efectos a ***ccw*** porque su efecto sobre la cara consistirá en hacerla visible de ambos lados, este campo es el llamado ***solid*** y por omisión toma el valor TRUE, lo que producirá una cara renderizada de un solo lado, mientras que si se lo pone a FALSE se podrán ver ambos lados.

Los puntos o vértices pertenecientes a la lista del campo ***coord*** deberán cumplir tres reglas para que la cara se represente de forma correcta. Estas consisten en que se necesitan un mínimo de tres puntos que no deben ser coincidentes para representar una cara; que los vértices deben estar en un mismo plano; y que ninguno de los lados del polígono definido debe intersecarse con el otro. De otra forma, los resultados que se obtengan son impredecibles al no estar definidos por el estándar.

Un caso particular es cuando el polígono que define la cara no es convexo, es decir que no todos sus ángulos interiores son menores que 180º. En este caso para la correcta representación se debe acudir al campo ***convex***, el cual por defecto vale FALSE, asignándole el valor TRUE. De esta manera la representación será la correcta.

En el Listado 14 se observa el código fuente para la construcción de un triángulo con el nodo **IndexedFaceSet**, nótese que el orden de los puntos define la figura en sentido contrario al de las agujas del reloj, en caso contrario debería especificarse el valor FALSE para el campo ***ccw***.

Para un ejemplo del uso del campo ***convex*** con el nodo **IndexedFaceSet** se modelarán dos figuras empleando el código del Listado 15. Nótese que a pesar de no ser necesario especificar el valor TRUE para la primera figura ya que es el valor por omisión, se ha hecho por motivos de claridad. Recuérdese también que al incluir un # en una línea de código, el resto de la misma será interpretado como un comentario.

```
#VRML V2.0 utf8
Shape {
   geometry IndexedFaceSet {
      coord Coordinate {
         point [1 0 0, 0 1 0, -1 0 0]
      }
      coordIndex [0,1,2,-1]
   }
}
```

Listado 14: Triángulo con el nodo IndexedFaceSet

```
#VRML V2.0 utf8
Shape {
   geometry IndexedFaceSet {
      coord Coordinate {
         point [2 0 0,  2 2 0,  1 2 0,  1 1 0,
                -1 1 0, -1 2 0, -2 2 0, -2 0 0]
      }
      coordIndex [0,1,2,3,4,5,6,7,-1]
      convex TRUE # VALOR POR DEFECTO
   }
}
Shape {
   geometry IndexedFaceSet {
      coord Coordinate {
         point [2 3 0,  2 5 0,  1 5 0,  1 4 0,
                -1 4 0, -1 5 0, -2 5 0, -2 3 0]
      }
      coordIndex [0,1,2,3,4,5,6,7,-1]
      convex FALSE
   }
}
```

Listado 15: Uso del campo convex en el nodo IndexedFaceSet

En la Figura 21 se pueden observar las figuras obtenidas con el código del Listado 14 y el del Listado 15. Nótese en la división derecha de esta figura en la que se presenta el cambio del comportamiento al poner a FALSE el campo ***convex*** (arriba) y dejarlo en su valor por defecto de TRUE (abajo). Debido a las características de la figura, para conseguir el correcto modelado es necesario el uso del valor FALSE en el campo ***convex***.

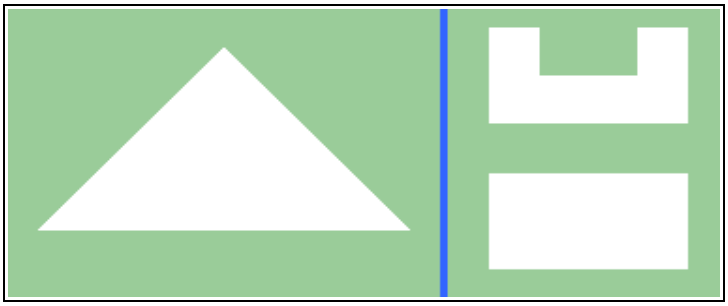


Figura 21: Triángulo y uso del campo convex del nodo IndexedFaceSet

Si el campo ***color*** es NULL (por omisión) la apariencia de las caras vendrá determinada por el campo ***apperance*** del nodo **Shape**. Si ***color*** no es NULL, se dan dos situaciones que dependerán del valor del campo ***colorPerVertex***:

Si el valor de ***colorPerVertex*** es TRUE, el cual es el predeterminado, los colores que se especifiquen en el campo ***color*** a través del constructor **Color** explicado en el Punto 3.2.6.3 se aplicarán a cada vértice en el orden dado por el campo ***colorIndex*** el cual sigue las mismas reglas que el campo ***coordIndex***, debiéndose terminar cada cara con el valor –1. Si la lista que se especifica para el campo ***colorIndex*** está vacía, o lo que es lo mismo no se incluye este campo dejándose entonces en su valor por defecto, se utilizará el orden dado por el campo ***coordIndex***.

Si el valor de ***colorPerVertex*** es FALSE los colores que se especifiquen en el campo ***color*** se aplicarán a cada cara en el orden dado por el campo ***colorIndex***. Si no se especifica este campo, el orden se tomará del campo ***coordIndex***.

En el Listado 16 se presenta el código fuente de una cara de sólido con cada vértice de un color distinto valiéndose del valor por omisión del campo ***colorPerVertex***. Los colores se definen utilizando el código RGB (véase la breve explicación en la Página 13 y el Punto 3.2.6.3 para más información sobre el nodo **Color**). Mientras que en el Listado 17 se define una figura de tres caras por medio de sendas caras de sólido, con un color distinto cada una, poniendo a FALSE el campo ***colorPerVertex***. Nótese la reutilización de los vértices comunes entre caras indicándose más de una vez su número de orden en la lista de índices asignada al campo ***coordIndex***, de esta manera se aprovecha cada punto ya definido reduciendo el tamaño del fichero fuente. Recuérdese que las comas son opcionales y que sólo se emplean para facilitar la lectura de las coordenadas.

```
#VRML V2.0 utf8
Shape {
   geometry IndexedFaceSet {
      coord Coordinate {
         point [-1 -1 0, 1 -1 0, 1 1 0, -1 1 0]
      }
      coordIndex [0,1,2,3,-1]
      color Color {
         color [1 0 0, 0 1 0, 0 0 1, 1 0 1]
      }
      colorIndex [0,1,2,3,-1]
```

```
      colorPerVertex TRUE # VALOR POR DEFECTO
   }
}
```

Listado 16: Uso del campo color con colorPerVertex valiendo TRUE

```
#VRML V2.0 utf8
Shape {
   geometry IndexedFaceSet {
      coord Coordinate {
         point [1 -1 1, 1 1 1, -1 1 1, -1 -1 1
                1 -1 -1, 1 1 -1]
      }
      coordIndex [0,1,2,3,-1
                  0 4 5 1 -1
                  2,1,5,-1]
      color Color {
         color [1 0 0, 0 1 0, 0 0 1]
      }
      colorIndex [0,1,2]
      colorPerVertex FALSE
   }
}
```

Listado 17: Uso del campo color con colorPerVertex valiendo FALSE

En la izquierda de la Figura 22 se presentan los resultados obtenidos con el Listado 16 y en la derecha los del Listado 17.

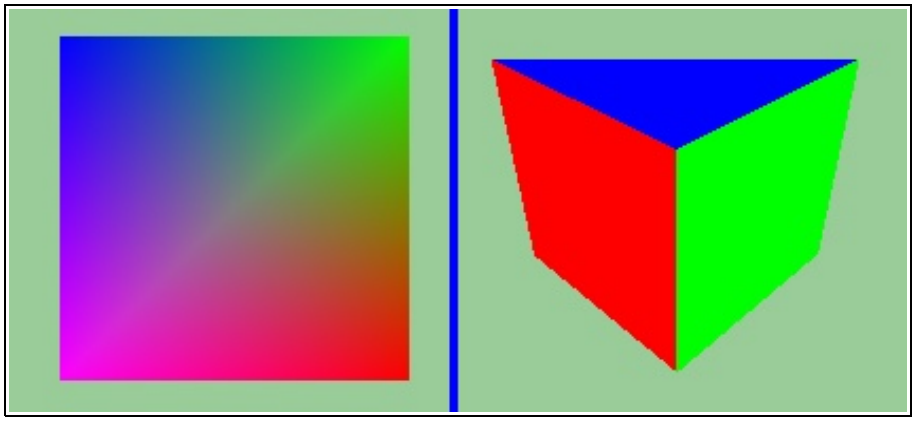


Figura 22: Uso de Colores con el nodo IndexedFaceSet

Si el campo ***normal*** es NULL (por defecto) el *plug-in* automáticamente generará las normales para cada uno de los vértices, en caso contrario el comportamiento será el mismo que para el caso del campo ***color***, teniendo en cuenta que el constructor de normales será el nodo **Normal** (véase el Punto 3.2.6.4). El orden de aplicación vendrá dado por el campo ***normalIndex*** y, si será respecto a los vértices o las caras, lo determinará el valor del campo ***normalPerVertex.***

El campo ***creaseAngle*** controlará cómo el *plug-in* VRML genera de forma automática a las normales. El ángulo que se especifique determinará el umbral a partir del cual cambia el estilo de sombreado y la forma en que se refleja la luz, variando así la suavidad de las aristas de las figuras.

En el Listado 18, se presenta el código fuente de dos caras situadas a 90º entre sí. Nótese que se ha utilizado el campo ***appearance*** del nodo **Shape**, definiendo el color blanco para todas las caras. Esto ha

sido necesario porque las normales operan sobre la forma en que se refleja la luz, por lo que era indispensable el uso del campo ***diffuseColor*** del constructor **Material**. Véase el Punto 3.3.1.1 para más datos sobre el nodo **Material**.

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {
      material Material {
         diffuseColor 1 1 1
      }
   }
   geometry IndexedFaceSet {
      coord Coordinate {
         point [1 0 0, 1 1 0, -1 1 0, -1 0 0
                1 0 -1, 1 1 -1]
      }
      coordIndex [0,1,2,3,-1
                  0,4,5,1,-1]
      creaseAngle 0 # VALOR POR DEFECTO
   }
}
```

Listado 18: creaseAngle por defecto en el nodo IndexedFaceSet

En Listado 19 presenta el código fuente que representa la misma entidad que el Listado 18, pero habiendo variado el ángulo a partir del cual cambia la forma en que el *plug-in* calcula las normales. Nótese que como los ángulos se expresan en radianes, el valor utilizado de 1.6 es sensiblemente superior a 1.5708 que se corresponde con los 90º sexagesimales, ángulo que forman las dos caras del ejemplo. Si hubiese sido menor que este valor no se apreciarían diferencias en el renderizado debidas a la utilización del campo ***creaseAngle***.

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {
      material Material {
         diffuseColor 1 1 1
      }
   }
   geometry IndexedFaceSet {
      coord Coordinate {
         point [1 0 0, 1 1 0, -1 1 0, -1 0 0
                1 0 -1, 1 1 -1]
      }
      coordIndex [0,1,2,3,-1
                  0,4,5,1,-1]
      creaseAngle 1.6
   }
}
```

Listado 19: Uso de creaseAngle en el nodo IndexedFaceSet

En la parte superior de la Figura 23 se observan los resultados obtenidos a través del Listado 18 a la izquierda y del Listado 19 a la derecha.

En el Listado 20 se puede observar el uso del campo ***normal*** definiendo el vector normal a cada vértice de las caras de sólido. Nótese que ***normalPerVertex*** adopta su valor por omisión, TRUE.

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {
      material Material {
         diffuseColor 1 1 1
      }
   }
   geometry IndexedFaceSet {
      coord Coordinate {
         point [1 0 0, 1 1 0, -1 1 0, -1 0 0
              1 0 -1, 1 1 -1]
      }
      coordIndex [0,1,2,3,-1
              0,4,5,1,-1]
      normal Normal {
         vector [.707 -.707 .707, .707 .707 .707,
                 .707 .707 .707,   .707 -.707 .707,
                -.707 -.707 .707, .707 -.707 -.707,
                 .707 .707 .707,   .707 .707 .707]
      }
      normalPerVertex TRUE # VALOR POR DEFECTO
   }
}
```

Listado 20: normalPerVertex por defecto en IndexedFaceSet

En el siguiente ejemplo ***normalPerVertex*** tiene un valor de FALSE, por lo que sólo será necesario especificar las normales para cada cara (véase el Listado 21).

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {
      material Material {
         diffuseColor 1 1 1
      }
   }
   geometry IndexedFaceSet {
      coord Coordinate {
         point [1 0 0, 1 1 0, -1 1 0, -1 0 0
                1 0 -1, 1 1 -1]
      }
      coordIndex [0,1,2,3,-1
                  0,4,5,1,-1]

      normal Normal {
         vector [.707 -.707 .707, .707 .707 .707]
      }
      creaseAngle 0
      normalPerVertex FALSE
   }
}
```

Listado 21: Uso de normalPerVertex en IndexedFaceSet

En la parte inferior de la Figura 23 se presentan las formas obtenidas con los listados precedentes, a la izquierda la producida por el Listado 20 y a la derecha la del Listado 21. Nótese las variaciones resultantes respecto al modelo original ubicado en la parte superior izquierda de la figura (el que se corresponde con el Listado 18).

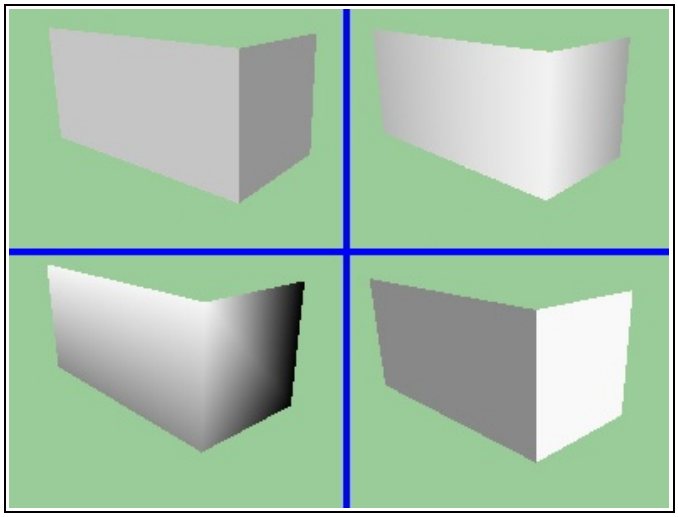


Figura 23: Uso de normales con el nodo IndexedFaceSet

El campo ***texCoord*** debe valer o NULL (por defecto) o bien contener el constructor **TextureCoordinate** (véase el Punto 3.2.6.2) mediante el cual se indica la correspondencia entre los vértices y las coordenadas de la textura que se aplicará a la cara. En los ejemplos se ha empleado el constructor **ImageTexture** para definir la textura con la cual se va a trabajar; este nodo será analizado en el Punto 3.3.1.2.

En el Listado 22 se presenta el código fuente que modela tres caras de sólido usando el nodo **IndexedFaceSet** utilizando para su aspecto una textura definida en el fichero *textura.gif*. El resultado obtenido se puede observar en la Figura 24 en el cuadrante superior izquierdo. Nótese que el plano sobre el que se aplica la textura es el de la cara de mayor tamaño de las definidas y que sobre las laterales se repite el patrón del perímetro de esta cara.

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {
      texture ImageTexture {url "textura.gif"}
   }
   geometry IndexedFaceSet {
      coord Coordinate {
         point [1 -1 1, 1 1 1, -1 1 1, -1 -1 1
                1 -1 –0.5, 1 1 –0.5, -1 1 –0.5]
      }
      coordIndex [0,1,2,3,-1
                  0 4 5 1 -1
                  2,1,5,6,-1]
   }
}
```

Listado 22: Caras con texturas creadas con el nodo IndexedFaceSet

La textura utilizada, contenida en el fichero *textura.gif*, se observa en la esquina inferior derecha de la Figura 24. La misma fue construida expresamente para poder comprobar el comportamiento que sigue VRML al distribuir una imagen sobre el objeto modelado.

El Listado 23 presenta la aplicación al mismo modelo del listado anterior de sólo una porción de la textura, para ello se hace uso del campo ***texCoord*** junto con el constructor **TextureCoordinate**. De esta manera se define el cuadrante correspondiente al número 3 de la textura para ser aplicado a la cara principal del modelo. Nótese que la textura se define dentro de un área de 1 unidad de lado, de forma que el punto (0,0) se corresponde con la esquina inferior izquierda de la textura y el punto (1,1) con la superior derecha. Véase en el cuadrante superior derecho de la Figura 24 el modelo obtenido.

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {
      texture ImageTexture {url "textura.gif"}
   }
   geometry IndexedFaceSet {
      coord Coordinate {
         point [1 -1 1, 1 1 1, -1 1 1, -1 -1 1
                1 -1 –0.5, 1 1 –0.5, -1 1 –0.5]
      }
      coordIndex [0,1,2,3,-1
                  0 4 5 1 -1
                  2,1,5,6,-1]
      texCoord TextureCoordinate {
         point [0.5 0, 0.5 0.5, 0 0.5, 0 0
                0.5 0, 0.5 0.5, 0 0.5, 0 0
                0.5 0, 0.5 0.5, 0 0.5, 0 0
                0.5 0, 0.5 0.5, 0 0.5, 0 0]
      }
   }
}
```

Listado 23: Uso del campo texCoord con IndexedFaceSet

Por último, para comprender el verdadero fin de utilizar partes de una textura, se aplicará al modelo de los ejemplos anteriores una región de textura por cada cara. El Listado 24 permite observar el código fuente para llevar a cabo esta labor.

```
#VRML V2.0 utf8
# CARA ANTERIOR
Shape {
   appearance Appearance {
      texture ImageTexture {url "textura.gif"}
   }
   geometry IndexedFaceSet {
      coord Coordinate {
         point [1 -1 1,  1 1 1, -1 1 1, -1 -1 1]
      }
      coordIndex [0,1,2,3,-1]
      texCoord TextureCoordinate {
         point [0.5 0, 0.5 0.5, 0 0.5, 0 0]
      }
   }
}
# CARA LATERAL
Shape {
   appearance Appearance {
      texture ImageTexture {url "textura.gif"}
   }
   geometry IndexedFaceSet {
      coord Coordinate {
         point [1 -1 1,  1 -1 –0.5, 1 1 –0.5, 1 1 1]
      }
      coordIndex [0,1,2,3,-1]
      texCoord TextureCoordinate {
```

```
            point [0.5 0, 1 0, 1 .5, 0.5 0.5]
         }
      }
   }
   # CARA SUPERIOR
   Shape {
      appearance Appearance {
         texture ImageTexture {url "textura.gif"}
      }
      geometry IndexedFaceSet {
         coord Coordinate {
            point [-1 1 1, 1 1 1, 1 1 –0.5, -1 1 –0.5]
         }
         coordIndex [0,1,2,3,-1]
         texCoord TextureCoordinate {
            point [.4 .4,.6 .4, .6 .6, .4 .6]
         }
      }
   }
```

Listado 24: Uso de zonas de textura distintas para cada cara

Primero se debe independizar cada cara para tratarlas por separado. A la cara anterior se le aplicará la región de textura que contiene el número tres, a la cara lateral se le aplicará la del número cuatro y a la cara superior, le será aplicada la región central de la textura que contiene el centro de la cruz.

Nótese que al definir por separado a cada cara se pierde la posibilidad de reutilizar los puntos en común debiendo ser repetidos en cada ocasión.

En la esquina inferior izquierda de la Figura 24 se observa el resultado obtenido con este último ejemplo.

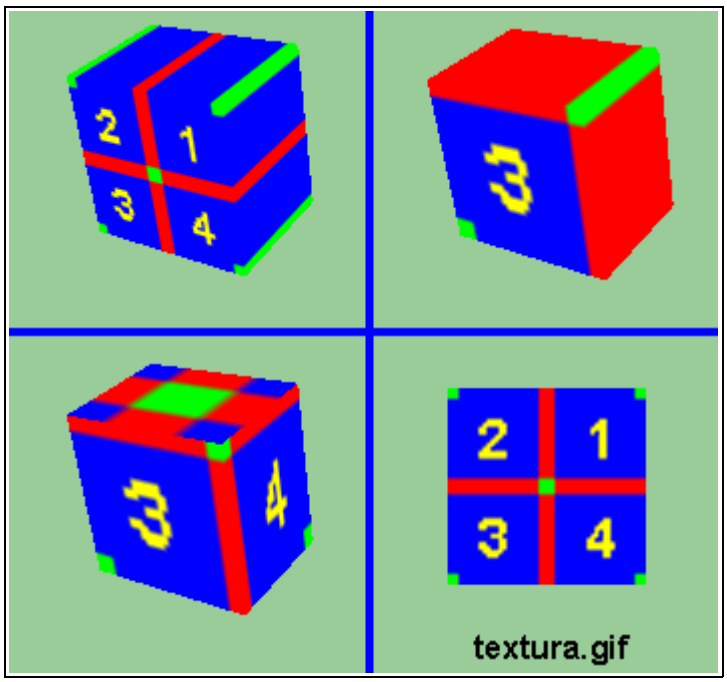


Figura 24: Uso de texturas con el nodo IndexedFaceSet

3.2.6.1 Coordinate

Sintaxis:

```
Coordinate {
      exposedField MFVec3f point  []      # (-∞,∞)
}
```

Este nodo se utiliza como constructor para generar una lista de puntos de tres dimensiones, satisfaciendo así las necesidades de los campos ***coordIndex*** de los nodos **IndexedFaceSet**, **IndexedLineSet** y **PointSet**.

3.2.6.2 TextureCoordinate

Sintaxis:

```
TextureCoordinate {
      exposedField MFVec2f point  []      # (-∞,∞)
}
```

Se emplea en los nodos **IndexedFaceSet** y **ElevationGrid** como constructor de coordenadas en dos dimensiones para especificar los valores requeridos por el campo ***texCoord***.

Su función consiste en hacer corresponder los vértices de una sección de una textura con los vértices de las caras que modelan a estos nodos. Las coordenadas de la textura toman al punto (0,0) como esquina inferior izquierda y al punto (1,1) como esquina superior derecha. Como se desprende de la sintaxis del nodo, se pueden utilizar valores mayores que 1 para las coordenadas, lo que redundará en una repetición de la textura. De esta manera una componente igual a 2 indicará que se repita dos veces la imagen en la dirección de ese eje (véase el Punto 3.3.1.2 para más información sobre las texturas).

3.2.6.3 Color

Sintaxis:

```
Color {
      exposedField MFColor color  []         # [0,1]
}
```

Este nodo se utiliza para construir una lista de colores que será empleada en el campo ***color*** de los nodos **IndexedFaceSet**, **IndexedLineSet**, **PointSet** y **ElevationGrid**.

Si además de usar este campo, se especifica dentro del campo ***geometry*** del nodo **Shape** algún tipo de material, los colores definidos por el nodo **Color** reemplazarán al valor del campo ***diffuseColor*** del material quedando el resto de las componentes del mismo inalteradas. Véase el Punto 3.3.1.1 para consultar los campos del nodo **Material**.

3.2.6.4 Normal

Sintaxis:

```
Normal {
      exposedField MFVec3f vector  []   # (-∞,∞)
}
```

Este nodo define una lista de vectores normalizados[24] para ser usada en el campo ***normal*** de los nodos **IndexedFaceSet** y **ElevationGrid**.

3.2.7 IndexedLineSet

Sintaxis:

```
IndexedLineSet {
      eventIn       MFInt32 set_colorIndex
      eventIn       MFInt32 set_coordIndex
      exposedField  SFNode  color             NULL
      exposedField  SFNode  coord             NULL
      field         MFInt32 colorIndex        []     # [-1, ∞)
      field         SFBool  colorPerVertex    TRUE
      field         MFInt32 coordIndex        []     # [-1, ∞)
}
```

Este nodo tiene un comportamiento similar al recientemente visto, **IndexedFaceSet**, sólo que en vez de modelar caras de sólido se modelarán segmentos en tres dimensiones. Los campos ***coord***, ***coordIndex***, ***color*** y ***colorIndex*** se comportan de igual forma que para el modelado de caras (véase el Punto 3.2.6), pero en este caso entre dos puntos sucesivos según el orden determinado por el índice, se trazará un segmento. Un –1 ahora significará el fin del segmento actual y el comienzo del siguiente.

```
#VRML V2.0 utf8
Shape {
   geometry IndexedLineSet {
      coord Coordinate {
         point [1 -1 1, 1 1 1, 0 0 1, -1 1 1, -1 -1 1]
      }
      coordIndex [0,1,2,3,4,-1]
      color Color {
         color [0 0 1, 0 1 0, 1 0 0, 1 0 1, 0 1 1]
      }
      colorPerVertex TRUE # VALOR POR DEFECTO
   }
}
Shape {
   geometry IndexedLineSet {
      coord Coordinate {
         point [1.5 -1 1, 1.5 1 1, 2 -1 1, 2 1 1,
                2.5 -1 1, 2.5 1 1]
      }
      coordIndex [0,1,-1, 2,3,-1, 4,5,-1]
      color Color {
         color [1 0 0, 0 1 0, 0 0 1]
      }
      colorPerVertex FALSE
   }
}
```

Listado 25: El nodo IndexedLineSet

[24] Con normalizado se quiere significar que el módulo del vector debe ser igual a uno.

Con el campo ***colorPerVertex,*** que por omisión vale TRUE, se determinará si los colores indicados en el campo ***color*** se aplicarán a cada vértice o a cada segmento en caso de valer FALSE.

En el Listado 25 se presenta el código fuente para modelar cuatro segmentos, el primero utiliza el coloreado por vértices mientras que para los tres restantes se utiliza el coloreado individual de todo el segmento asignando al campo ***colorPerVertex*** el valor FALSE y obteniendo el efecto que se observa en la Figura 25. Nótese que a los tres segmentos verticales se los ha modelado dentro de un mismo nodo **Shape** haciendo uso de las coordenadas indicadas en el campo ***coord***, separando cada segmento por un –1 en la definición de los índices en el campo ***colorIndex***.

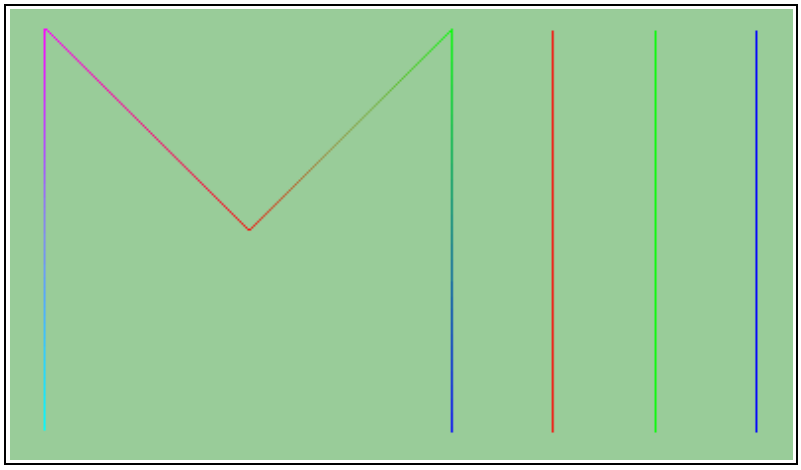


Figura 25: El nodo IndexedLineSet

Obsérvese la gradación de color en los cuatro primeros segmentos. Como último apunte sobre este nodo, destacar que si no se especifica el campo ***color*** se tomará el contenido del campo ***emissiveColor*** del material indicado en el campo ***appearance*** del nodo **Shape** o bien el material predeterminado.

3.2.8 PointSet

Sintaxis:

```
PointSet {
      exposedField  SFNode  color      NULL
      exposedField  SFNode  coord      NULL
}
```

El nodo **PointSet** modela una serie de puntos ubicados en las coordenadas de tres dimensiones indicadas en su campo ***coord*** a través del nodo constructor **Coordinate,** asignándole a todos el color ***emissiveColor*** definido en el campo **Apperance** del nodo **Shape** o bien individualmente a través del campo ***color*** por medio del nodo **Color**.

El Listado 26 presenta un ejemplo de uso de este nodo y en la Figura 26 se puede observar el resultado obtenido.

```
#VRML V2.0 utf8
Shape {
   geometry PointSet {
      coord Coordinate {
         point [1 -1 1, 1 1 1, 0 0 1, -1 1 1, -1 -1 1]
      }
      color Color {
         color [0 0 0, 0 0 1, 0 0 0, 0 0 1, 0 0 0]
      }
   }
}
```

Listado 26: El nodo PointSet

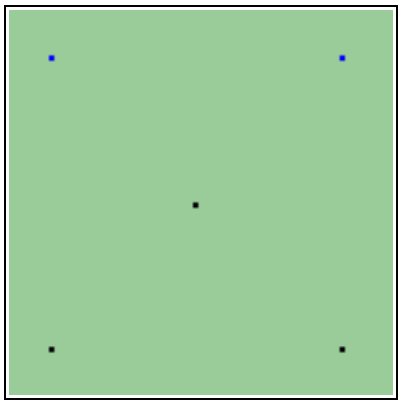


Figura 26: El nodo PointSet

3.2.9 ElevationGrid

Sintaxis:

```
ElevationGrid {
      eventIn      MFFloat  set_height
      exposedField SFNode   color             NULL
      exposedField SFNode   normal            NULL
      exposedField SFNode   texCoord          NULL
      field        MFFloat  height            []      # (-∞,∞)
      field        SFBool   ccw               TRUE
      field        SFBool   colorPerVertex    TRUE
      field        SFFloat  creaseAngle       0       # [0, ∞]
      field        SFBool   normalPerVertex   TRUE
      field        SFBool   solid             TRUE
      field        SFInt32  xDimension        0       # [0, ∞)
      field        SFFloat  xSpacing          1.0     # (0, ∞)
      field        SFInt32  zDimension        0       # [0, ∞)
      field        SFFloat  zSpacing          1.0     # (0, ∞)
}
```

El nodo **ElevationGrid** describe una superficie dividida en una malla de rectángulos (ver Figura 27). Las elevaciones que se corresponden con el eje ***y*** de coordenadas en el sistema local se definirán en el campo **height** el cual es una lista de valores en coma flotante que se corresponderán con cada sección de la malla.

Esta malla vendrá definida en el plano ***xz*** a través de los campos ***xDimension*** y ***zDimension*** que determinarán su tamaño en puntos y por ***xSpacing*** y ***zSpacing*** que indicarán el tamaño en metros para los rectángulos que forman la malla y que serán todos de dimensiones iguales.

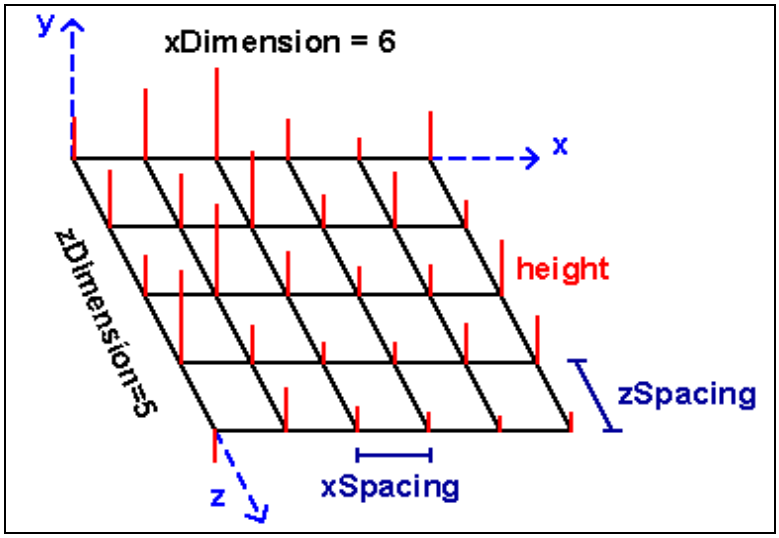


Figura 27: El nodo ElevationGrid

En el Listado 27 se presenta un ejemplo de uso del nodo **ElevationGrid** construido con una malla de 5 x 5 puntos compuesta por cuadrados de 1 metro de lado. Nótese que el número de puntos de elevación especificados en el campo ***height*** se corresponde con la cantidad de puntos en la malla: 25 = 5 x 5. En la izquierda de la Figura 28 se observa la superficie obtenida.

Téngase en cuenta que el sombreado se aprecia gracias a la inclusión de la componente de ***diffuseColor*** en el material utilizado para el campo ***appearance*** del nodo **Shape**, evitando que tome su valor por defecto.

```
#VRML V2.0 utf8
Shape {
      appearance Appearance {
            material Material {
         diffuseColor 1 1 1
      }
   }
   geometry ElevationGrid {
      height [1 0 0 0 -1, 0 0 1 0 0, 0 1 1 1 0,
              0 1 2 1 0, 0 3 5 3 0]
      xDimension 5
      xSpacing 1
      zDimension 5
      zSpacing 1
   }
}
```

Listado 27: El nodo ElevationGrid

Al igual que para el nodo **IndexedFaceSet** analizado en el Punto 3.2.6, los campos ***ccw*** y ***solid*** controlan qué lado de la superficie se renderiza o si se hará en ambos lados. Para los ejemplos se han dejado con su valor por omisión igual a TRUE.

Las normales se tratan también de igual forma que para el nodo **IndexedFaceSet** tomando cada elemento de la malla como una cara individual. De esta manera si se especifica el campo ***normal*** y el campo ***normalPerVertex*** vale TRUE (por defecto) se deberán incluir dentro del constructor **Normal** los

vectores de normales para cada uno de los puntos que definen la alturas de la superficie. Mientras que si el campo ***normalPerVertex*** toma el valor FALSE los vectores de normales necesarios serán los correspondientes a cada una de las caras.

Si el campo ***normal*** vale NULL, o lo que es lo mismo no se especifica, el *plug-in* calculará por si mismo las normales utilizando el contenido del campo ***creaseAngle*** para determinar la suavidad de la transición entre las caras de la malla.

En el Listado 28 se presenta la misma superficie que en el Listado 27 donde se ha utilizado un valor de 1.5 radianes para el campo ***creaseAngle***. En la derecha de la Figura 28 se pueden observar los cambios que introduce esta variación.

```
#VRML V2.0 utf8
Shape {
      appearance Appearance {
            material Material {
         diffuseColor 1 1 1
      }
   }
   geometry ElevationGrid {
      height [1 0 0 0 -1, 0 0 1 0 0, 0 1 1 1 0,
              0 1 2 1 0, 0 3 5 3 0]
      xDimension 5
      xSpacing 1
      zDimension 5
      zSpacing 1
      creaseAngle 1.5
   }
}
```

Listado 28: Uso del campo creaseAngle del nodo ElevationGrid

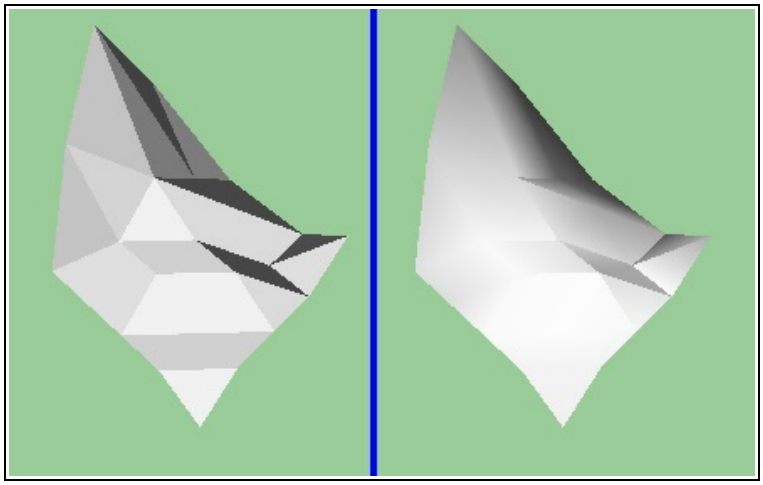


Figura 28: Uso del campo creaseAngle del nodo ElevationGrid

Para permitir la comparación de resultados se utilizará como en otras ocasiones la misma superficie para la mayoría de los ejemplos referentes a este nodo, la cual se ha elegido de forma que se presenten diversas variaciones de alturas para potenciar las diferencias del comportamiento frente a los cambios de los valores de los campos.

Abordando ahora el uso de colores, el método es similar al utilizado para el nodo **IndexedFaceSet**. De esta manera si el valor del campo ***colorPerVertex*** es TRUE, se especificará en el campo ***color*** mediante el constructor **Color**, un número de colores igual al número de alturas definidas en el campo ***height***. Un ejemplo de la aplicación de estos valores se presenta en el Listado 29 en donde se van rotando los colores para cada vértice tal como se puede observar en la izquierda de la Figura 29.

```
#VRML V2.0 utf8
Shape {
      appearance Appearance {
            material Material {
         diffuseColor 1 1 1
      }
   }
   geometry ElevationGrid {
      height [1 0 0 0 -1, 0 0 1 0 0, 0 1 1 1 0,
            0 1 2 1 0, 0 3 5 3 0]
      xDimension 5
      xSpacing 1
      zDimension 5
      zSpacing 1
      colorPerVertex TRUE # VALOR POR DEFECTO
      color Color {
         color [0 0 0, 0 0 1, 0 1 0, 1 0 0, 1 1 0,
                1 0 1, 0 1 1, 1 1 1, 0 1 1, 1 0 1,
                1 1 0, 1 0 0, 0 1 0, 0 0 1, 0 0 0,
                0 0 1, 0 1 0, 1 0 0, 1 1 0, 1 0 1,
                0 1 1, 1 1 1, 0 1 1, 1 0 1, 1 1 0]
      }
   }
}
```

Listado 29: Uso del campo colorPerVertex por defecto del nodo ElevationGrid

Ahora bien, si el valor del campo ***colorPerVertex*** no es el predeterminado, es decir si vale FALSE, entonces los colores definidos en el campo ***color*** serán utilizados para cada cara de la malla. En el Listado 30 se observa el código fuente que representa a la superficie de los ejemplos pero esta vez cambiando los colores de cada cara. Véase en la zona central de la Figura 29 el aspecto de la superficie al colorear cada cara con un color diferente.

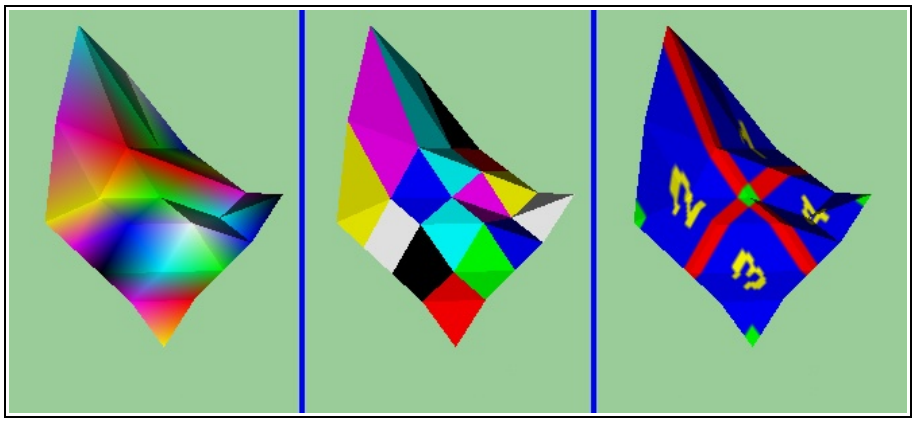


Figura 29: Aplicación de colores y texturas al nodo ElevationGrid

```
#VRML V2.0 utf8
Shape {
      appearance Appearance {
            material Material {diffuseColor 1 1 1}
   }
   geometry ElevationGrid {
      height [1 0 0 0 -1, 0 0 1 0 0, 0 1 1 1 0,
              0 1 2 1 0, 0 3 5 3 0]
      xDimension 5
      xSpacing 1
      zDimension 5
      zSpacing 1
      colorPerVertex FALSE
      color Color {
         color [1 1 1, 0 0 1, 0 1 0, 1 0 0,
                1 1 0, 1 0 1, 0 1 1, 0 0 0,
                1 0 0, 0 1 1, 0 0 1, 1 1 1,
                0 0 0, 0 1 1, 1 0 1, 1 1 0]
      }
   }
}
```

Listado 30: colorPerVertex igual al FALSE del nodo ElevationGrid

El campo ***texCoord***, igual que con el nodo **IndexedFaceSet,** define la porción de textura a aplicar pero esta vez se deberá indicar la correspondencia entre los puntos de la textura con cada uno de los puntos de la matriz de alturas. Obsérvese en el Listado 31 el uso del campo ***texCoord*** y los resultados obtenidos que se pueden ver en la derecha de la Figura 29.

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {
      texture ImageTexture {
         url "textura.gif"
      }
      material Material {
         diffuseColor 1 1 1
      }
   }
   geometry ElevationGrid {
      height [1 0 0 0 -1, 0 0 1 0 0, 0 1 1 1 0,
              0 1 2 1 0, 0 3 5 3 0]
      xDimension 5
      xSpacing 1
      zDimension 5
      zSpacing 1
      texCoord TextureCoordinate {
         point [0 0,    0 0.25,    0 0.5,    0 0.75,    0 1,
                0.25 0, 0.25 0.25, 0.25 0.5, 0.25 .75,  0.25 1,
                0.5 0,  0.5 0.25,  0.5 0.5,  0.5 .75,   0.5 1,
                0.75 0, 0.75 0.25, 0.75 0.5, 0.75 0.75, 0.75 1,
                1 0,    1 0.25,    1 0.5,    1 0.75,    1 1]
      }
   }
}
```

Listado 31: Uso de texturas con el nodo ElevationGrid

3.2.10 Extrusion

Sintaxis:

```
Extrusion {
      eventIn MFVec2f    set_crossSection
      eventIn MFRotation set_orientation
      eventIn MFVec2f    set_scale
```

```
      eventIn MFVec3f    set_spine
      field   SFBool     beginCap         TRUE
      field   SFBool     ccw              TRUE
      field   SFBool     convex           TRUE
      field   SFFloat    creaseAngle      0                # [0,∞)
      field   MFVec2f    crossSection     [ 1 1, 1 -1, -1 -1, -1 1, 1 1 ]   # (-∞,∞)
      field   SFBool     endCap           TRUE
      field   MFRotation orientation      0 0 1 0          # [-1,1],(-∞,∞)
      field   MFVec2f    scale            1 1              # (0,∞)
      field   SFBool     solid            TRUE
      field   MFVec3f    spine            [ 0 0 0, 0 1 0 ] # (-∞,∞)
}
```

El nodo **Extrusion** se utiliza para crear superficies por medio de una figura en el plano que luego será rotada y/o escalada a lo largo de una curva en el espacio. Con ello se pueden representar sólidos de revolución y una gran variedad de formas cilíndricas.

El campo ***crossSection*** alberga la lista de puntos de dos dimensiones que representará a una figura que será la sección con la cual se construirá el sólido. Luego esta figura recorrerá la curva en el espacio determinada por los puntos de la lista del campo ***spine***, modelando en su trayecto la superficie deseada.

El campo ***ccw*** determinará si se va a modelar el lado interno o el externo de la figura y el campo ***solid*** al tomar el valor FALSE, ocasionará que se modelen ambos lados de la misma lo que aumenta los cálculos que debe realizar el *plug-in* por lo que se evitará su uso siempre que sea posible. Por otro lado, el campo ***convex*** será necesario que valga FALSE cuando la figura de la sección (campo ***crossSection***) no sea un polígono convexo (véase el uso de ***convex*** junto con el nodo **IndexedFaceSet** en el Punto 3.2.6).

El Listado 32 presenta el código fuente de una superficie generada con un nodo **Extrusion**. Nótese que debido a la forma en que se construyó la cara de la sección ha sido necesario poner a FALSE los campos ***ccw*** y ***convex***.

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {
      material Material {
         diffuseColor 1 1 1
      }
   }
   geometry Extrusion {
      crossSection [0.5 0, 2 1, 1 1, 1 2, 0 4, -1 2, -1 1, -0.5 0,
                    -1 -1, -1 -2, 0 -4, 1 -2, 1 -1, 2 -1, 0.5 0]
      spine [7 -5 0, 4 -5 0, 2 -4 0, 1 -3 0, 0 -1 0,
             0 1 0, 1 3 0, 2 4 0, 4 5 0, 7 5 0]
      ccw FALSE
      convex FALSE
   }
}
```

Listado 32: El nodo Extrusion

En la izquierda de la Figura 30 se observa el modelo conseguido y en la parte central se puede ver como afecta el uso del campo ***creaseAngle*** a la forma en que se sombrean las caras, dando un aspecto mayor de suavidad.

En el Listado 33 se presenta el uso del campo ***creaseAngle*** que, como se ha visto ya con otras primitivas, indica a partir de qué ángulo se empiezan a suavizar las transiciones entre caras.

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {
      material Material {
         diffuseColor 1 1 1
      }
   }
   geometry Extrusion {
      crossSection [0.5 0, 2 1, 1 1, 1 2, 0 4, -1 2, -1 1, -0.5 0,
                    -1 -1, -1 -2, 0 -4, 1 -2, 1 -1, 2 -1, 0.5 0]
      spine [7 -5 0, 4 -5 0, 2 -4 0, 1 -3 0, 0 -1 0,
             0 1 0, 1 3 0, 2 4 0, 4 5 0, 7 5 0]
      ccw FALSE
      convex FALSE
      creaseAngle 0.75
   }
}
```

Listado 33: Uso del campo creaseAngle con el nodo Extrusion

El campo ***scale*** especifica la transformación que sufrirá la sección en cada punto indicado por el campo ***spine***. De esta manera ***scale*** deberá contener tantos valores como puntos tiene el campo ***spine*** o bien, como caso particular, ***scale*** constará de un sólo valor de escala indicando la escala para toda la superficie.

En el Listado 34 se presenta el uso del campo ***scale*** y en la derecha de la Figura 30 se observa el cambio de aspecto que sufre la figura frente a esta transformación. Nótese que se ha mantenido el valor del campo ***creaseAngle*** igual a 0.75 radianes como en el listado anterior para obtener una superficie más suave.

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {
      material Material {
         diffuseColor 1 1 1
      }
   }
   geometry Extrusion {
      crossSection [0.5 0, 2 1, 1 1, 1 2, 0 4, -1 2, -1 1, -0.5 0,
                    -1 -1, -1 -2, 0 -4, 1 -2, 1 -1, 2 -1, 0.5 0]
      spine [7 -5 0, 4 -5 0, 2 -4 0, 1 -3 0, 0 -1 0,
             0 1 0, 1 3 0, 2 4 0, 4 5 0, 7 5 0]
      ccw FALSE
      convex FALSE
      creaseAngle 0.75
      scale [.1 .1, .5 .5, .5 .5, .5 .5, 1.2 1.2,
           1.2 1.2, .5 .5, .25 .25, .5 .5, 1 1]
   }
}
```

Listado 34: Uso del campo scale con el nodo Extrusion

El campo ***orientation*** permite definir una rotación particular para cada punto que forma el campo ***spine***. Nuevamente como en el caso del campo ***scale*** si sólo se especifica un valor, éste se aplica a toda la

primitiva. En caso contrario se deberán especificar un valor de rotación (eje y ángulo) para cada punto del campo ***spine***.

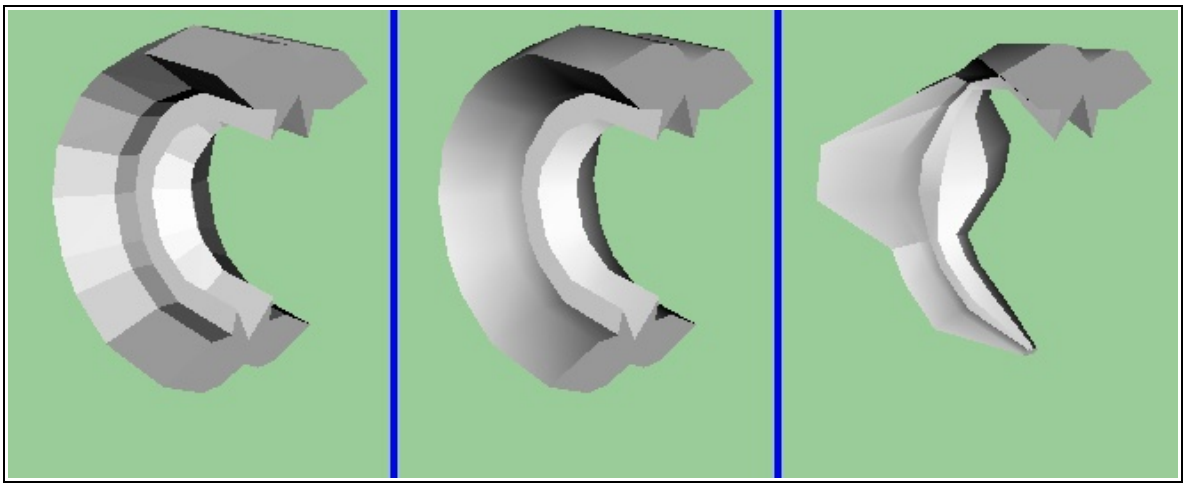


Figura 30: Uso de los campos creaseAngle y scale con el nodo Extrusion

En el Listado 35 se puede observar un nodo **Extrusion** que hace uso del campo ***orientation*** para rotar en cinco pasos su sección, obteniéndose los resultados que se presentan en la Figura 31 en donde se puede observar la superficie sin la transformación y su aspecto luego de ser rotada.

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {
      material Material {
         diffuseColor 1 1 1
      }
   }
   geometry Extrusion {
      crossSection [ -2 -.1, -2 .1, 2 .1, 2 -.1, -2 -.1]
      spine [0 -4 0, 0 -2 0, 0 0 0, 0 2 0, 0 4 0]
      orientation [0 1 0 0, 0 1 0 0.3927,
                   0 1 0 0.7854, 0 1 0 1.1781, 0 1 0 1.5708]
      creaseAngle 2.5
   }
}
```

Listado 35: Uso del campo orientation con el nodo Extrusion

Los campos ***beginCap*** y ***endCap*** controlan si serán modeladas las caras de comienzo y final de la superficie, de igual forma que se ha visto con el nodo **Cylinder** en el Punto 3.2.4.

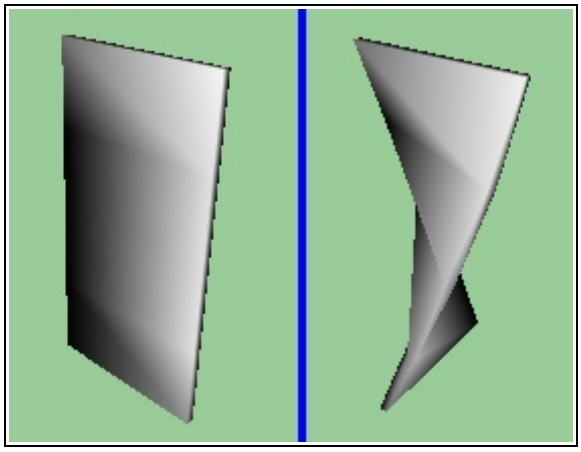


Figura 31: Uso del campo orientation con el nodo Extrusion

3.2.11 Text

Sintaxis:

```
Text {
      exposedField  MFString string    []
      exposedField  SFNode   fontStyle NULL
      exposedField  MFFloat  length    []      # [0,∞)
      exposedField  SFFloat  maxExtent 0.0     # [0,∞)
}
```

El nodo **Text** permite representar textos en el mundo virtual. Para ello dispone del campo ***string*** que permite especificar una lista de cadenas de caracteres (hasta 100 cadenas de 100 caracteres cada una) que serán modeladas utilizando el estilo indicado por el campo ***fontStyle*** analizado posteriormente en el Punto 3.2.11.1.

Mediante el campo ***length*** se especifica la longitud deseada para cada una de las cadenas del campo ***string***, de manera que si una de las cadenas tiene una longitud mayor que la indicada se reducirá el espaciado entre sus caracteres o se disminuirá el tamaño de cada letra, ocurriendo lo contrario en el caso de que la cadena sea de una longitud menor.

El campo ***maxExtent*** limitará la longitud de todas las cadenas que conforman la lista ***string*** siguiendo el mismo método que para el campo ***length***. Si alguna de las cadenas no posee una longitud mayor que la especificada en este campo la misma no sufrirá ninguna modificación.

En el Listado 36 se presentan tres ejemplos de uso del nodo **Text** con las mismas dos cadenas. En el primer caso sin restricciones, indicando las longitudes particulares de cada una en el segundo caso y restringiendo la máxima longitud en el tercero.

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {
            material Material {diffuseColor 0 1 0}
      }
   geometry Text {
      string ["VRML97","El Nodo Text"]
   }
}
Transform {
   translation 0 3 0
   children
      Shape {
            appearance Appearance {
                  material Material {diffuseColor 1 0 0}
            }
            geometry Text {
               string ["VRML97","El Nodo Text"]
               length [7,6]
            }
      }
}
Transform {
   translation 0 6 0
   children
      Shape {
            appearance Appearance {
                  material Material {diffuseColor 0 0 0}
            }
            geometry Text {
               string ["VRML97","El Nodo Text"]
               maxExtent 3.5
            }
      }
}
```

Listado 36: El nodo Text

En la Figura 32 se observan los resultados obtenidos con el listado precedente.

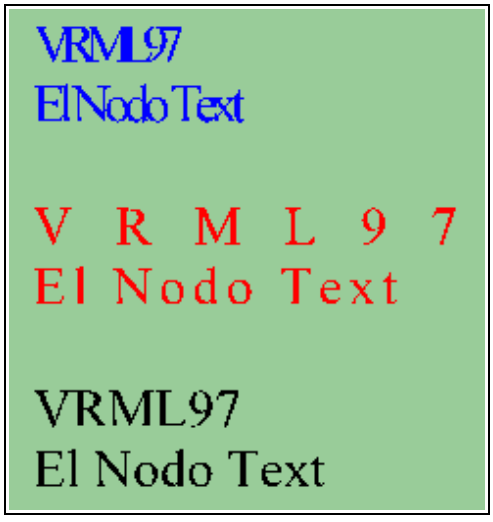


Figura 32: El nodo Text

3.2.11.1 FontStyle

Sintaxis:

```
FontStyle {
      field MFString family       "SERIF"
      field SFBool   horizontal   TRUE
      field MFString justify      "BEGIN"
      field SFString language     ""
      field SFBool   leftToRight  TRUE
      field SFFloat  size         1.0          # (0,∞)
      field SFFloat  spacing      1.0          # [0,∞)
      field SFString style        "PLAIN"
      field SFBool   topToBottom  TRUE
}
```

El nodo **FontStyle** se utiliza en el campo **fontStyle** del nodo **Text** permitiendo definir estilo, lenguaje y dirección de escritura para los caracteres que se emplearán en la construcción de las cadenas representadas dentro del mundo virtual por el nodo **Text**.

El campo ***family*** indica el tipo de letra utilizado pudiendo variar entre plataformas y entre *plug-ins*. El estándar asegura como mínimo la existencia de los tipos **"SERIF"** (por defecto), **"SANS"** y **"TYPEWRITER"**.

El campo ***size*** hace referencia a la altura promedio de los caracteres, mientras que el campo ***spacing*** se refiere a la separación entre líneas de texto sólo aplicable cuando se tiene más de una cadena en el campo ***string*** del nodo **Text**.

El campo ***horizontal*** indica la dirección de la escritura, mientras que los campos ***leftToRight*** y ***topToBottom*** indican de qué forma se van sucediendo los caracteres que componen la cadena.

El campo ***justify*** se encarga de especificar la alineación de los caracteres. Podrá tomar los valores **"BEGIN"** o **"FIRST"** (por omisión) indicando una alineación a la izquierda en cada línea, **"MIDDLE"** que indica una alineación centrada en el sentido de la escritura y **"END"** que produce un alineamiento a la derecha. Dependiendo de los valores de los campos ***leftToRight***, ***topToBottom*** y ***horizontal*** el comportamiento del nodo **Text** respecto al campo ***justify*** se verá alterado para adecuarse al nuevo estilo de escritura.

El campo ***style*** especifica el estilo en que se representarán las letras de la cadena. Los valores posibles son: **"PLAIN"** (por defecto), **"BOLD"** e **"ITALIC"**, correspondiéndose con los estilos normal, negrita y cursiva respectivamente.

Por último el campo ***language*** se utiliza para indicar el lenguaje en que se encuentra el texto de la cadena de caracteres. El uso de este campo es necesario debido a los múltiples idiomas que permite utilizar VRML a través de la codificación UTF-8.

El Listado 37 presenta cuatro cadenas de caracteres modeladas con distintos estilos de texto. Los resultados obtenidos se pueden observar en la Figura 33.

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {
      material Material {diffuseColor 0 0 1}
   }
   geometry Text {
      string "VRML97"
   }
}
Transform {
   translation 0 2 0
   children
      Shape {
         appearance Appearance {
            material Material {diffuseColor 0 0 1}
         }
         geometry Text {
            string "VRML97"
            fontStyle FontStyle {
               family "TYPEWRITER"
               size 2
               style "ITALIC"
            }
         }
      }
}
Transform {
   translation 0 4 0
   children
      Shape {
         appearance Appearance {
            material Material {diffuseColor 0 0 1}
         }
         geometry Text {
            string "VRML97"
            fontStyle FontStyle {
               family "SANS"
               style "BOLD"
               leftToRight FALSE
            }
         }
      }
}
Transform {
   translation 6 0 0
   children
      Shape {
         appearance Appearance {
            material Material {diffuseColor 0 0 1}
         }
         geometry Text {
            string "VRML97"
            fontStyle FontStyle {
               family "TYPEWRITER"
               horizontal FALSE
            }
         }
      }
}
```

Listado 37: El nodo FontStyle

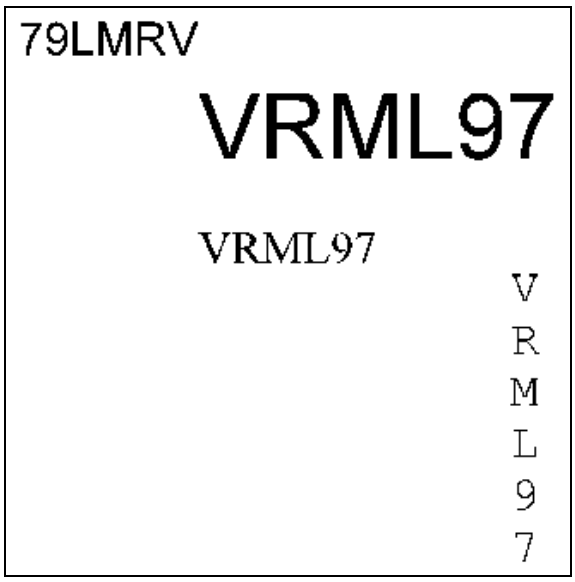


Figura 33: El nodo FontStyle

3.3 Materiales y texturas

Habiendo culminado con los nodos que modelan formas, figuras y superficies, pasamos a tratar el aspecto que presentarán estas primitivas. Para ello se utilizará el constructor **Appearance** que será el encargado de determinar el aspecto de una primitiva.

3.3.1 Appearance

Sintaxis:

```
Appearance {
      exposedField SFNode material          NULL
      exposedField SFNode texture           NULL
      exposedField SFNode textureTransform  NULL
}
```

El nodo **Appearance** especifica las características del aspecto de una primitiva. Se le referencia desde el campo ***appearance*** de un nodo **Shape** y afectará al contenido del campo ***geometry*** del mismo nodo.

El campo ***material***, si no es NULL, va seguido del nodo constructor **Material** (véase el Punto 3.3.1.1) el cual definirá las distintas componentes de colores que conformarán el material a utilizar para el modelado de la primitiva. Si ***material*** es NULL no se utilizarán los efectos de iluminación modelándose el objeto con el color blanco sin sombras ni variación de tonalidades dificultando la interpretación de su forma.

El campo ***texture,*** si no es NULL, contendrá a un nodo **ImageTexture** (Punto 3.3.1.2), **MovieTexture** (Punto 3.3.1.3) o **PixelTexture** (Punto 3.3.1.4), que serán analizados en sus respectivos apartados y que determinarán el mapa de bits que será aplicado sobre la primitiva que se esté modelando.

Por último el campo ***textureTransform***, si no toma el valor NULL, irá seguido del nodo **TextureTransform** analizado en el Punto 3.3.1.5 que definirá una serie de transformaciones que podrán ser aplicadas a la textura definida en el campo ***texture*** del nodo **Appearance**.

3.3.1.1 Material

Sintaxis:

```
Material {
      exposedField SFFloat ambientIntensity  0.2         # [0,1]
      exposedField SFColor diffuseColor      0.8 0.8 0.8 # [0,1]
      exposedField SFColor emissiveColor     0 0 0       # [0,1]
      exposedField SFFloat shininess         0.2         # [0,1]
      exposedField SFColor specularColor     0 0 0       # [0,1]
      exposedField SFFloat transparency      0           # [0,1]
}
```

Este nodo es el encargado de construir un material haciendo uso de sus campos los que determinan tres tipos de colores y tres características de comportamiento frente a la luz.

Los colores vienen definidos por tres componentes (RGB) correspondientes a los colores Rojo (Red), Verde (Green) y Azul (Blue). Según la proporción que se utilice de cada uno se obtendrá una tonalidad distinta. Nótese que cada componente varía entre 0 y 1, mientras que normalmente en las aplicaciones de dibujo se utiliza un rango de 0 a 255. Aplicando regla de tres se obtiene la Fórmula 1 que permite la conversión de un rango de colores al otro.

$$componente_{(0-1)} = \frac{componente_{(0-255)}}{255}$$

Fórmula 1: Conversión de componentes RGB

El campo ***diffuseColor*** hace referencia al color propio del objeto el cual será reflejado al ser iluminado por alguna de las luces del modelo. Si no hay luz alguna (la del *avatar* inclusive) que ilumine a una entidad modelada sólo con ***diffuseColor***, esta no será visible al no haber ningún reflejo.

El campo ***emissiveColor*** se refiere al color interno que emitirá la entidad independizándose de una fuente de luz externa para ser visible. Si su valor es distinto del negro (RGB igual a 0 0 0), este toma precedencia frente a los otros colores. Nótese que esta no es una fuente de luz y que sólo afecta a la primitiva modelada.

En la Figura 34 se aprecia la diferencia en el aspecto de un cubo modelado sólo con ***diffuseColor*** a la izquierda y sólo con ***emissiveColor*** a la derecha. En ambos casos se ha utilizado el color blanco (RGB igual a 1 1 1).

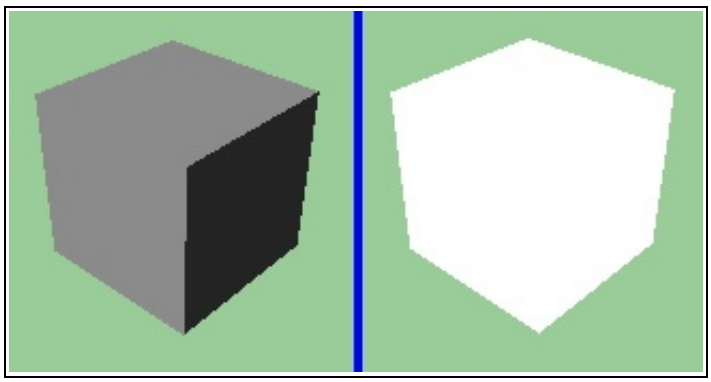


Figura 34: Uso de diffuseColor y emissiveColor del nodo Material

El campo ***specularColor*** hace referencia al color que refleja la primitiva cuando el ángulo entre la fuente de luz y la superficie se encuentra próximo al ángulo entre la superficie y el observador. El grado de reflejo viene dado por el campo ***shininess*** de forma que para valores cercanos al cero produce un efecto similar al plástico y para valores cercanos a uno los efectos se asemejan al del metal.

En la Figura 35 se observan dos cilindros: el de la izquierda sólo con ***diffuseColor*** azul y el de la derecha con ***specularColor*** blanco y un valor de ***shininess*** igual a 0.8.

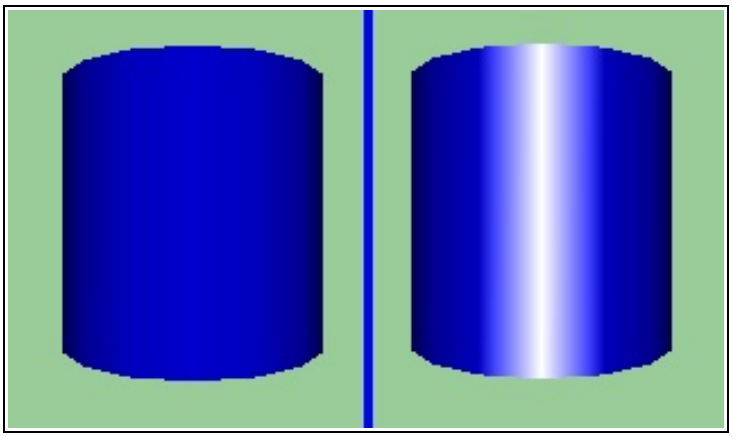


Figura 35: Uso de specularColor y shininess del nodo Material

El campo ***ambientIntensity*** indica qué proporción de luz ambiente refleja la primitiva (véase en el Punto 3.4 las componentes de la luz en VRML). Para los ejemplos se ha utilizado el valor predeterminado para este campo el cual es igual a 0.2.

Por último el campo ***transparency*** indicará el grado de transparencia del objeto variando desde 0 (totalmente opaco) hasta 1 (totalmente transparente). En la Figura 36 se presentan cuatro conos modelados con valores del campo ***transparency*** igual a 0, 0.3, 0.6 y 0.8 respectivamente. Nótese la inclusión de una caja azul y opaca que atraviesa el cono para facilitar la visualización del grado de transparencia empleado.

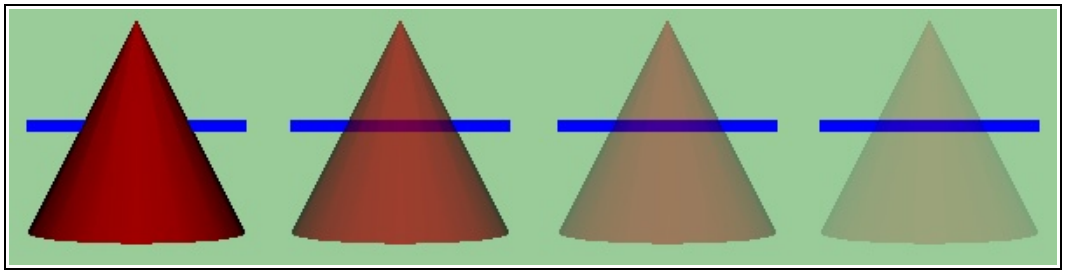


Figura 36: Uso del campo transparency del nodo Material

3.3.1.2 ImageTexture

Sintaxis:

```
ImageTexture {
      exposedField MFString url      []
      field        SFBool   repeatS TRUE
      field        SFBool   repeatT TRUE
}
```

El nodo **ImageTexture** define una textura que se corresponde con un fichero cargado a través del campo ***url*** el cual es una lista que puede estar vacía y de esa manera se desactivan las texturas, o puede contener una o más (hasta 10) referencias a ficheros de imágenes *.GIF*, *.JPG*, *.PNG* ó *.CGM*. Si el primer fichero especificado no se encuentra se intentará cargar el segundo y así sucesivamente hasta poder emplear uno de la lista o hasta que ésta se agote[25].

Una textura representa un plano en dos dimensiones con dos ejes ***t*** y ***s*** (véase la Figura 37) de esta manera con los campos ***repeatS*** y ***repeatT*** se hará referencia a la forma de aplicación de la misma sobre las caras de un sólido. Si estos campos valen TRUE (por defecto) la textura se repetirá más allá de las líneas ***s*** = 1 y ***t*** = 1, mientras que si su valor es igual a FALSE sólo se aplicará una copia y lo que se expandirá será el contorno de la misma.

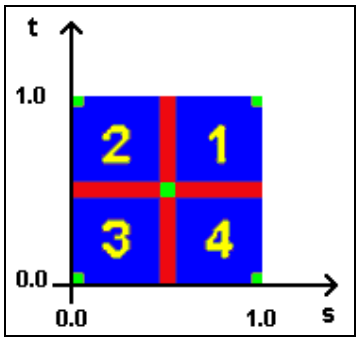


Figura 37: Sistema de coordenadas de una textura

[25] Este comportamiento para el campo ***url*** será común en todos los nodos que contengan un campo de este nombre. Como las referencias pueden ser locales, o direcciones de Internet haciendo uso de recursos remotos, su principal cometido es prever la no disponibilidad momentánea o permanente de alguno de ellos. Si a pesar de todo, una referencia de este tipo no consigue ubicar el recurso solicitado se continuará con la carga del mundo virtual no tomándose como una situación errónea pero echándose en falta luego dentro del modelo. Se podrán utilizar los protocolos http, ftp y file.

Como aplicación de este nodo se presenta el Listado 38 en el cual la textura de ejemplo que se está utilizando en este libro se aplica a una caja modelada con el nodo **Box**. Nótese que para las otras primitivas la porción de código para el campo ***appearance*** del nodo **Shape** es idéntica a la del ejemplo y que sólo variará el contenido del campo ***geometry*** del nodo.

Además de la textura se ha utilizado un material que tiene definido su campo ***diffuseColor*** de esta forma se evita el empleo del material por omisión que no permite la visualización de las sombras y desvirtuaría los ejemplos.

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {
      material Material {diffuseColor 1 1 1}
      texture ImageTexture {url "textura.gif"}
   }
   geometry Box { }
}
```

Listado 38: Aplicación de texturas a un nodo Box

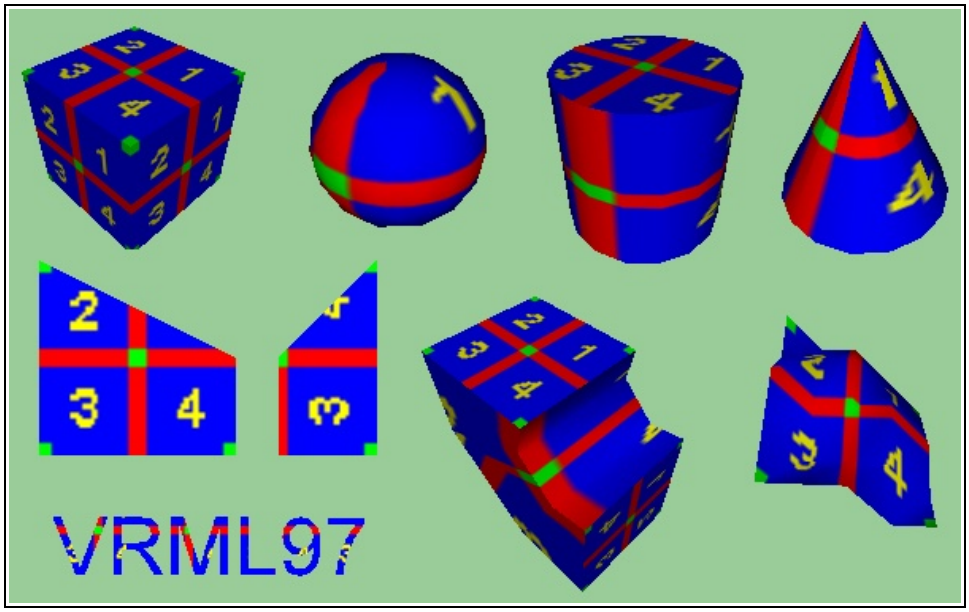


Figura 38: Aplicación de texturas a las primitivas

En la Figura 38 se observan las distintas primitivas estudiadas y la forma en que se aplican las texturas según el tipo de su superficie. Nótese que para las caras de sólido el eje **s** de la textura se alinea con la dimensión mayor de la figura.

3.3.1.3 MovieTexture

Sintaxis:

```
MovieTexture {
      exposedField SFBool   loop            FALSE
      exposedField SFFloat  speed           1.0       # (-∞,∞)
      exposedField SFTime   startTime       0         # (-∞,∞)
```

```
      exposedField SFTime   stopTime          0        # (-∞,∞)
      exposedField MFString url               []
      field        SFBool   repeatS           TRUE
      field        SFBool   repeatT           TRUE
      eventOut     SFTime   duration_changed
      eventOut     SFBool   isActive
}
```

Este nodo permite aplicar, a modo de textura, una película en formato MPG[26]. Se lo puede referenciar desde el campo ***texture*** del nodo **Appearance** visualizándose el film como textura y desde el campo ***source*** del nodo **Sound** (véase el Punto 3.5.1) para reproducir el sonido de la película.

El campo ***url*** permite indicar la ruta del fichero a reproducir permitiéndose especificar más de uno. De esta manera igual que con el nodo **ImageTexture**, se intentará cargar el primero. Si no se encuentra intentará con el segundo, etc. Véase la Nota 25 para más información sobre el campo ***url***.

Los campos ***startTime*** y ***stopTime*** hacen referencia a la hora de comienzo y de fin de la reproducción[27]. Cuando se alcanza la hora indicada por el campo ***startTime*** la película comienza a reproducirse y al mismo tiempo el evento de salida ***isActive*** adopta el valor TRUE. Al terminarse la película o al alcanzarse la hora indicada por el campo ***stopTime*** se termina la reproducción y el evento de salida ***isActive*** pasa ahora a valer FALSE.

Si el campo ***loop*** toma el valor TRUE el comportamiento variará sensiblemente. Si ***stopTime*** es menor que ***startTime*** la reproducción será cíclica e ininterrumpida. Si por el contrario ***stopTime*** es mayor que ***startTime*** y a la vez mayor que la duración de la película, cuando finalice la primera reproducción se volverá a comenzar desde el principio hasta que se alcance la hora indicada por el campo ***stopTime***.

El campo ***speed*** determinará la velocidad de reproducción del vídeo y si se le asigna un valor negativo la película se reproducirá en sentido inverso. Aunque esta característica aún no se encuentra implementada en todos los *plug-ins*.

Al cargarse la película el evento de salida ***duration_changed*** toma el valor de la duración total del film, mientras que cuando aún no ha sido cargada su valor será igual a –1.

En el Listado 39 se presenta un ejemplo del uso del nodo **MovieTexture**. Nótese que se ha adoptado como ***stopTime*** una hora anterior a ***startTime*** e igual a –1 para asegurar el comienzo de la reproducción una vez que el fichero sea cargado y que ésta sea continua debido al uso del valor TRUE para el campo ***loop***.

[26] Adaptación a extensiones de tres caracteres de M*PEG* (Moving Pictures Expert Group – Grupo de Expertos en Imágenes en Movimiento), que hace referencia al formato de audio y vídeo comprimido.

[27] La hora en VRML consiste en un número en coma flotante de doble precisión representando los segundos transcurridos desde la hora 0:00:00 GMT del 1 de Enero de 1970. Un valor negativo se interpreta como que ha ocurrido con anterioridad a 1970.

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {
      material Material {diffuseColor 1 1 1}
      texture MovieTexture {
         url "video.mpg"
         startTime 0
         stopTime -1
         loop TRUE
      }
   }
   geometry Box {size 1 1 .5}
}
```

Listado 39: Aplicación del nodo MovieTexture al nodo Box

En la Figura 39 se observa la aplicación del nodo **MovieTexture** a distintos tipos de primitivas.

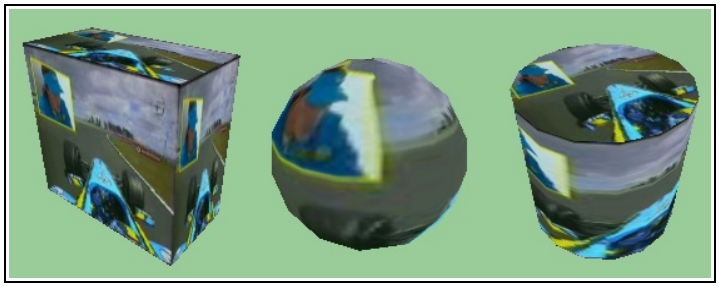


Figura 39: Aplicación del nodo MovieTexture a distintas primitivas

Por último, los campos ***repeatS*** y ***repeatT*** son los encargados de determinar el comportamiento de la textura respecto a la repetición en ambos ejes el cual sigue los mismos patrones que para el nodo **ImageTexture** (véase el Punto 3.3.1.2).

3.3.1.4 PixelTexture

Sintaxis:

```
PixelTexture {
      exposedField SFImage  image      0 0 0
      field        SFBool   repeatS    TRUE
      field        SFBool   repeatT    TRUE
}
```

Este nodo define una textura por medio de su campo ***image*** el cual contiene a una matriz de píxeles con la información de color para cada uno de ellos más una cabecera que consistirá en tres valores: el ancho y el alto de la imagen y además el tipo de información de color, véase la Tabla 2.

En el Listado 40 se presenta un ejemplo de uso del nodo **PixelTexture** a través del código fuente que se corresponde con el ejemplo del tipo 3, presentado en la Tabla 2.

Tipo	Significado	Ejemplo
1	Una componente de color, que se corresponde con 256 tonalidades de grises.	2 2 **1** 0x00 0x44 0x88 0xFF
2	Una componentes de color, que se corresponde con 256 tonalidades de grises y una segunda componente para el grado de opacidad (canal alfa).	2 2 **2** 0xFF66 0x0066 0x0066 0xFF66
3	Tres componentes de color, que se corresponden con los tres niveles RGB respectivos.	2 2 **3** 0xFF0000 0x00FF00 0x0000FF 0xFF00FF
4	Tres componentes de color, que se corresponden con los tres niveles RGB respectivos y una cuarta componente para el grado de opacidad (canal alfa).	2 2 **4** 0xFF000066 0x00FF0066 0x0000FF66 0xFF00FF66

Tabla 2: Estructura de SFImage

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {
      texture PixelTexture {
         image 2 2 3 0xFF0000 0x00FF00 0x0000FF 0xFF00FF
      }
   }
   geometry Box {size 1 1 .5}
}
```

Listado 40: Uso del nodo PixelTexture (ejemplo del Tipo 3)

En la Figura 40 se observan los cuatro ejemplos de la Tabla 2 aplicados a una cara de un cubo. Obsérvese los niveles de transparencia para el segundo y cuarto caso.

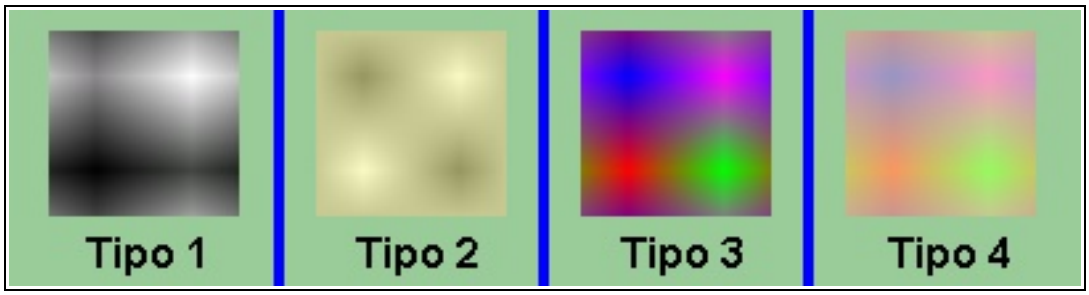


Figura 40: El nodo PixelTexture

Para finalizar, los campos ***repeatS*** y ***repeatT*** determinarán la repetición de la textura respecto a ambos ejes de igual modo que para los nodos **ImageTexture** y **MovieTexture** (véanse los Puntos 3.3.1.2 y 3.3.1.3 respectivamente).

3.3.1.5 TextureTransform

Sintaxis:

```
TextureTransform {
      exposedField SFVec2f center      0 0     # (-∞,∞)
      exposedField SFFloat rotation    0       # (-∞,∞)
      exposedField SFVec2f scale       1 1     # (-∞,∞)
      exposedField SFVec2f translation 0 0     # (-∞,∞)
}
```

Este nodo realiza transformaciones en dos dimensiones a una textura. Para ello se lo referencia desde el campo ***textureTransform*** del nodo **Appearance**, el cual también deberá incluir una textura a través de su campo ***texture*** para que pueda ser modificada de lo contrario las transformaciones serán ignoradas.

El campo ***scale*** define la cantidad de copias que se realizan en ambos ejes siempre y cuando los campos ***repeatS*** y ***repeatT*** del nodo empleado como textura (**ImageTexture**, **MovieTexture** ó **PixelTexture**) tomen su valor por defecto de TRUE.

El Listado 41 presenta un ejemplo de uso del campo ***scale*** del nodo **TextureTransform** en el cual se aplican 25 copias (5 x 5) del fichero *textura.gif* al nodo **Sphere** ya empleado en el ejemplo de la Figura 38 y en el cual se le había aplicado una sola copia.

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {
      material Material {diffuseColor 1 1 1}
      texture ImageTexture {url "textura.gif"}
      textureTransform TextureTransform {scale 5 5}
   }
   geometry Sphere { }
}
```

Listado 41: Uso del campo scale del nodo TextureTransform

En la Figura 41 se observa la esfera del Listado 41, el cubo del Listado 39 al cual se le aplican esta vez 4 copias (2 x 2) de la textura de película, la cara del cubo del Listado 40 ahora con 16 copias (4 x 4) de la textura creada píxel a píxel y por último un cilindro particular que debería tener aplicadas 4 copias de la textura, pero al tener su textura los campos ***repeatS*** y ***repeatT*** valiendo FALSE, se le ha aplicado sólo una vez.

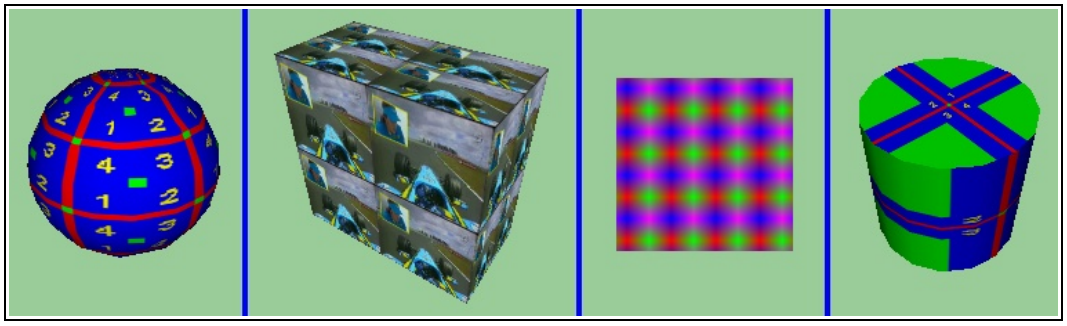


Figura 41: Uso del campo scale del nodo TextureTransform

El campo ***translation*** provoca un desplazamiento de la textura a través de los ejes ***s*** y/o **t**. En el Listado 42 se presenta el código fuente de un cubo modelado por el nodo **Box** al que se le aplica la textura del fichero *textura.gif* y luego se la traslada media unidad en ambos ejes.

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {
      material Material {diffuseColor 1 1 1}
      texture ImageTexture {url "textura.gif"}
      textureTransform TextureTransform {translation .5 .5}
   }
   geometry Box { }
}
```

Listado 42: Uso del campo translation del nodo TextureTransform

El campo ***rotation*** cumple la función de rotar la textura. Como esta rotación es en el plano, no hace falta indicar el eje como en las rotaciones en el espacio (véase el Punto 3.6.2) siendo suficiente especificar el ángulo en radianes solamente. En el Listado 43 se observa la aplicación del campo ***rotation*** a la textura utilizada en una caja de tamaño predeterminado.

El campo ***center*** se empleará para fijar el centro de la rotación y del escalado de las texturas. En el Listado 44 se presenta la rotación de la textura aplicada al nodo **Box** del ejemplo anterior pero esta vez colocando el centro de forma apropiada.

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {
      material Material {diffuseColor 1 1 1}
      texture ImageTexture {url "textura.gif"}
      textureTransform TextureTransform {
         rotation .7854 # PI / 4
      }
   }
   geometry Box { }
}
```

Listado 43: Uso del campo rotation del nodo TextureTransform

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {
      material Material {diffuseColor 1 1 1}
      texture ImageTexture {url "textura.gif"}
      textureTransform TextureTransform {
         rotation .7854 # PI / 4
         center -.5 -.5 # CENTRA LA TEXTURA PARA LA ROTACION
      }
   }
   geometry Box { }
}
```

Listado 44: Uso del campo center del nodo TextureTransform

En la Figura 42 se pueden observar los resultados de los listados analizados previamente en este punto comparándose (de izquierda a derecha) el cubo sin ninguna transformación, el cubo con la textura trasladada, el cubo con la textura rotada y el cubo con la textura centrada y rotada.

Para finalizar con este nodo queda aclarar el orden en que las transformaciones se aplican a una textura: En primer lugar, se aplicará la traslación seguida por la rotación sobre el punto central previamente fijado o sobre el punto por defecto (0,0) y, por último, se realizará el escalado también respecto al punto central.

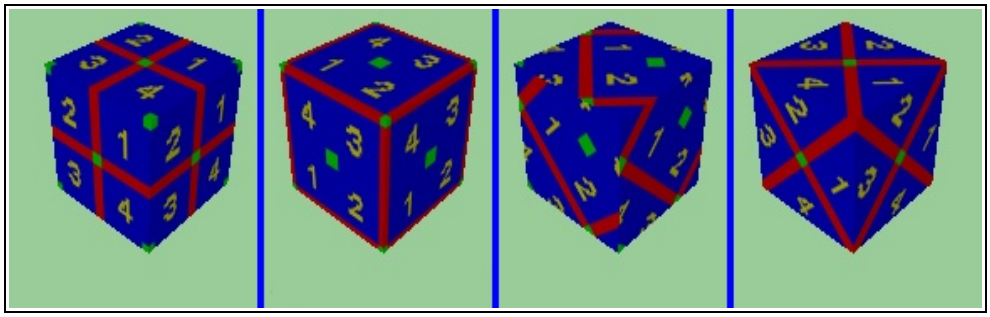


Figura 42: Uso de los campos translation, rotation y center del nodo TextureTransform

3.4 Iluminación

Además de la luz propia del *avatar* existe la posibilidad, y es en general lo recomendable, de dotar al modelo de sus propias fuente de luz. El modelo de iluminación que provee VRML consiste en una aproximación del mundo real. De esta manera, cada primitiva modelada recibe la suma de las componentes lumínicas correspondiente a la iluminación directa y a la ambiental.

Para ello, cada fuente de luz provee de los correspondientes campos especificando el color, la intensidad de luz directa y la intensidad de luz ambiental, que en conjunto definirán la características propias de cada fuente de luz.

Una restricción importante impuesta por la especificación de VRML es que se limitan a un máximo de ocho luces simultáneas dentro de un mundo virtual.

3.4.1 DirectionalLight

Sintaxis:

```
DirectionalLight {
      exposedField SFFloat ambientIntensity  0        # [0,1]
      exposedField SFColor color             1 1 1    # [0,1]
      exposedField SFVec3f direction         0 0 -1   # (-∞,∞)
      exposedField SFFloat intensity         1        # [0,1]
      exposedField SFBool  on                TRUE
}
```

Este nodo define una luz direccional que afecta a todos los nodos que se encuentran dentro de su mismo grupo y a sus hijos dentro del fichero VRML (véanse los nodos de agrupación en el Punto 3.6). El efecto obtenido es el de una luz distante, como podría ser la emitida por el sol, con todos sus rayos paralelos y que no se atenúan con la distancia.

Sus campos ***intensity*** y ***ambientIntensity*** controlan la intensidad de ambas componentes, la primera respecto a la luz directa, y la segunda respecto al aporte que realiza a la luz ambiental de la escena.

El campo ***color*** especifica las componentes RGB de la luz emitida y el campo ***on*** se refiere a si la luz estará encendida (TRUE) o apagada (FALSE). Este campo será empleado para encender y apagar las luces a voluntad por medio de sensores y/o *scripts*.

Por último, el campo ***direction*** hace referencia a las tres componentes de un vector que determinarán la dirección y el sentido en el que emanan los rayos de luz.

En el Listado 45 se aprecia el código que utiliza el nodo **DirectionalLight** para iluminar a un cubo modelado con el nodo **Box**, el cual se ha modelado con un material de color blanco y con un ***ambientIntensity*** de 0.5. De esta manera se consigue que la componente ***ambientIntensity*** de la fuente de luz afecte a la iluminación del cubo ya que de otra forma sólo incidiría la componente ***intensity*** de la misma.

```
#VRML V2.0 utf8
DirectionalLight {
   color 0 1 1
   ambientIntensity 1
   intensity 0.5
   direction 1 -1 -1
}
Shape {
   appearance Appearance {
      material Material {
         diffuseColor 1 1 1
         ambientIntensity 0.5
      }
   }
   geometry Box { }
}
```

Listado 45: El nodo DirectionalLight

El resultado obtenido para distintos valores del campo ***ambientIntensity*** del nodo **DirectionalLight** se observa en la Figura 43. Nótese que respecto al Listado 45 sólo se ha modificado el valor del campo mencionado.

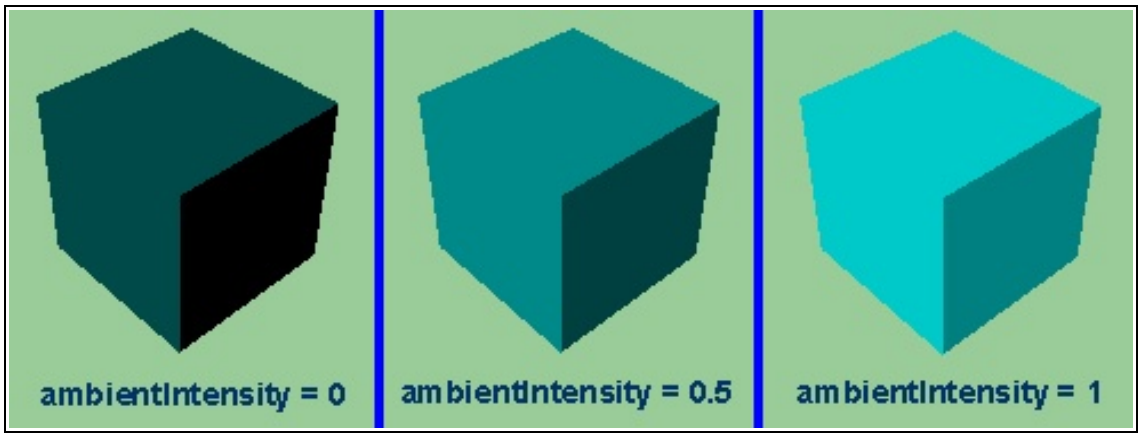


Figura 43: Diferentes niveles de ambientIntensity con el nodo DirectionalLight

3.4.2 PointLight

Sintaxis:

```
PointLight {
      exposedField SFFloat ambientIntensity  0       # [0,1]
      exposedField SFVec3f attenuation       1 0 0   # [0,∞)
      exposedField SFColor color             1 1 1   # [0,1]
      exposedField SFFloat intensity         1       # [0,1]
      exposedField SFVec3f location          0 0 0   # (-∞,∞)
      exposedField SFBool  on                TRUE
      exposedField SFFloat radius            100     # [0,∞)
}
```

Este nodo modela un punto de luz ubicado en las coordenadas locales determinadas por el campo ***location***, y con las características particulares que vienen dadas por el resto de sus campos.

Al igual que con el nodo precedente, **DirectionalLight**, los campos ***ambientIntensity***, ***intensity*** y ***color*** definen las componentes de iluminación ambiental, iluminación directa y el color de la luz. Lo mismo ocurre con el campo ***on*** que cumple la función de controlar el encendido y el apagado.

El campo ***radius*** representa el alcance de la luz en metros, definiendo una esfera de luz de forma que todo sólido ubicado dentro de su volumen sea iluminado.

Por medio del campo ***attenuation*** se variará la forma en que va disminuyendo la intensidad luminosa, a medida que aumenta la distancia al punto de origen de la luz definido por el campo ***location***. Las tres componentes representarán los coeficientes de un polinomio de segundo grado aplicándose para el cálculo la Fórmula 2 que definirá el coeficiente de atenuación. Nótese que el valor máximo para este coeficiente es 1, evitando así que la atenuación se convierta en amplificación lo que no sería correcto.

$$coeficiente = \frac{1}{\max\left(1, \text{attenuation}[0] + \text{radius} \times \text{attenuation}[1] + \text{radius} \times \text{attenuation}[2]^2\right)}$$

Fórmula 2: Cálculo de la atenuación para el nodo PointLight

En el Listado 46 se observa como ejemplo la porción de código en donde se define un punto de luz centrado en el origen de coordenadas, con un radio de cobertura de 5 metros y de color cian.

```
#VRML V2.0 utf8
PointLight {
   color 0 1 1
   ambientIntensity 0.5
   intensity 1
   radius 5
}
# A CONTINUACION SE DEFINEN LAS PRIMITIVAS
```

Listado 46: El nodo PointLight

En la Figura 44 se presenta el nodo **PointLight** del listado anterior ubicado en el punto indicado por la pequeña esfera oscura, la cual no está iluminada al tener en su interior la fuente de luz. Luego ésta

alcanza a los conos superiores pero no a los inferiores por encontrarse a una distancia mayor de la fuente de luz. En el fondo, una caja permite apreciar la atenuación progresiva de la luz que está siendo emitida.

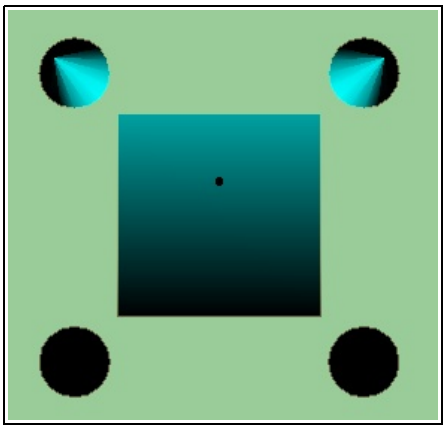


Figura 44: El nodo PointLight

3.4.3 SpotLight

Sintaxis:

```
SpotLight {
      exposedField SFFloat ambientIntensity  0          # [0,1]
      exposedField SFVec3f attenuation       1 0 0      # [0,∞)
      exposedField SFFloat beamWidth         1.570796   # (0,PI/2]
      exposedField SFColor color             1 1 1      # [0,1]
      exposedField SFFloat cutOffAngle       0.785398   # (0,PI/2]
      exposedField SFVec3f direction         0 0 -1     # (-∞,∞)
      exposedField SFFloat intensity         1          # [0,1]
      exposedField SFVec3f location          0 0 0      # (-∞,∞)
      exposedField SFBool  on                TRUE
      exposedField SFFloat radius            100        # [0,∞)
}
```

Este nodo define un punto de luz al igual que el nodo anterior, **PointLight**, pero en este caso la luz emitida no es omnidireccional, sino que la misma está dirigida hacia un punto del espacio. Además es posible definir el cono de penumbra donde la intensidad luminosa irá decayendo progresivamente.

El campo ***location*** sitúa el origen de la luz en el espacio. Luego por medio del campo ***direction***, se define el vector tridimensional que indicará la dirección y el sentido en que la luz se va a propagar hasta alcanzar la distancia indicada por el campo ***radius***.

Por otro lado, el campo ***beamWidth*** definirá el ángulo que compone al cono de luz donde la intensidad viene indicada por el campo ***intensity***, mientras que el campo ***cutOffAngle*** será el encargado de definir un segundo cono de mayor tamaño, en el cual el nivel de iluminación irá decayendo hasta desaparecer (véase la Figura 45).

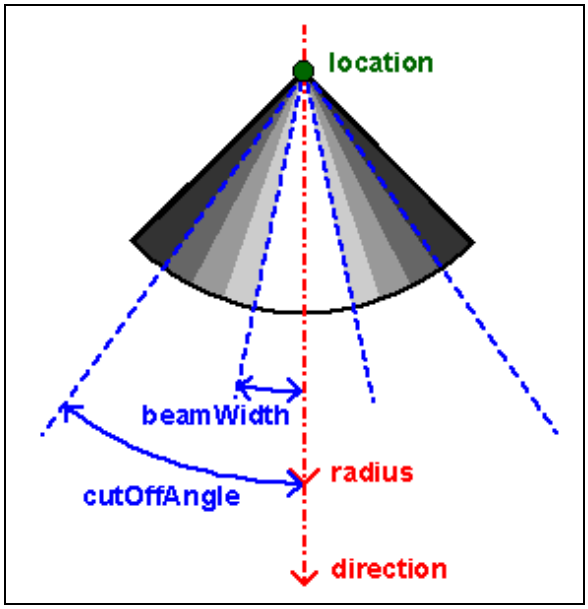


Figura 45: El nodo SpotLight

Los campos ***color***, ***intensity***, ***ambientIntensity*** y ***attenuation*** terminan de definir las características de este nodo, y su funcionalidad es la misma que para el nodo **PointLight** analizado recientemente en el Punto 3.4.2.

Queda por analizar el campo ***on***, el cual también obedece al comportamiento ya explicado para las otras dos luces, encendiendo o apagando la misma.

Nuevamente, en este caso las atenuaciones implementadas por algunos *plug-ins* son de progresión constante, haciendo caso omiso a la lineal y a la exponencial (véase la Fórmula 2).

Como ejemplo se va a modelar tres conos blancos iluminando el de un extremo con un nodo **SpotLight** de luz roja, el del opuesto con uno de luz azul, mientras que el cono del centro recibirá la luz proveniente de la zona de penumbra de ambas luces.

En el Listado 47 se presenta la parte del código del ejemplo en donde se definen las dos luces utilizando sendos nodos **SpotLight**.

```
#VRML V2.0 utf8
# LUZ ROJA
SpotLight {
   ambientIntensity 1
   color 1 0 0
   beamWidth .2
   cutOffAngle 1
   direction -1 0 0
   intensity 1
   location 4 0 2.5
}
# LUZ AZUL
SpotLight {
   ambientIntensity .5
   color 0 0 1
   beamWidth .15
   cutOffAngle .2
```

```
   direction -.577 -.577 .577
   intensity 1
   location 2 3 -7
}

# A CONTINUACION SE DEFINEN LOS TRES CONOS
```

Listado 47: El nodo SpotLight

En la Figura 46 se pueden ver los efectos obtenidos. Nótese que los conos blancos reflejan el color de la luz con la que son iluminados, no pudiéndose discernir a simple vista el color del material con que han sido modelados.

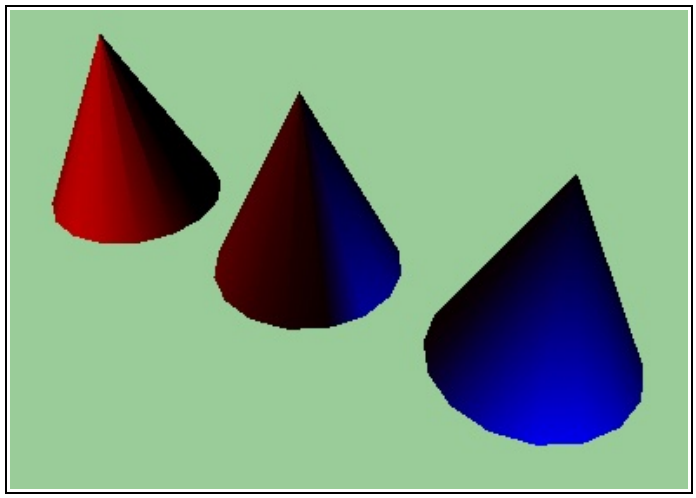


Figura 46: Tres conos blancos iluminados por dos nodos SpotLight

A continuación se presenta un ejemplo curioso en el que se refleja el alto coste computacional del nodo **SpotLight**. Se pretende iluminar una cara de sólido mediante una luz blanca modelada con un nodo **SpotLight**, de manera que pueda observarse la proyección de ambos conos de luz sobre la cara de sólido, la que al estar perpendicular al eje central de la luz presentará un aspecto circular.

Para ello se ha escrito el código del Listado 48 definiendo una luz blanca orientada hacia la cara de sólido. Pero al entrar en el mundo virtual, los efectos obtenidos fueron los que se observan en la Figura 47 a). Nótese que la cara no recibe luz alguna, aunque el problema en realidad radica en que los vértices no reciben luz, debido a que la intención era obtener un circulo iluminado en el centro de la cara de sólido.

El motivo es que VRML, para calcular la iluminación de una cara emplea la luz que incide en sus vértices, pero como en este caso los mismos quedan fuera del cono de luz e incluso del de penumbra, se obtiene una cara totalmente a oscuras.

La solución sería dibujar varias caras pequeñas formando una malla, de modo que mediante sus vértices, obtener los diferentes niveles de intensidad luminosa. Cuantas más caras se dibujen, más precisa será la iluminación.

```
#VRML V2.0 utf8
SpotLight {
   ambientIntensity .5
   color 1 1 1
   beamWidth .2
   cutOffAngle 1.5
   direction 0 -1 0
   intensity 1
   location 2.25 4 2.25
}
Shape {
   appearance Appearance {
      material Material {
         diffuseColor 1 1 1
         ambientIntensity .5
      }
   }
   geometry IndexedFaceSet {
            coord Coordinate {
                  point [0 0 0, 0 0 5, 5 0 5, 5 0 0]
            }
            coordIndex [0,1,2,3,-1]
      }
}
```

Listado 48: Cara de sólido iluminada por el nodo SpotLight

Para el ejemplo se modelarán 100 caras de sólidos en una malla de 10 x 10, pero en vez de usar el nodo **IndexedFaceSet** que requeriría demasiados puntos para definir cada cara, se utilizará el nodo **ElevationGrid** que nos provee de la malla y especificando todas las alturas igual a cero se obtendrá el mismo resultado.

El código fuente se presenta en el Listado 49 y el resultado obtenido en la Figura 47 b). Nótese que ahora sí se observan los efectos de la iluminación, pero las transiciones entre las caras son demasiados abruptas. Recordando los campos del nodo **ElevationGrid** (véase el Punto 3.2.9), más precisamente el campo ***creaseAngle*** el cual era el encargado de fijar el ángulo a partir del cual se suavizan las transiciones entre caras. Aplicando entonces esta técnica, asignando un valor de 1.5 radianes al campo ***creaseAngle***, se obtiene el resultado que se aprecia en la Figura 47 c) en donde se ha obtenido, por fin, la representación de ambos conos de luz sobre la superficie.

```
#VRML V2.0 utf8
SpotLight {
   ambientIntensity .5
   color 1 1 1
   beamWidth .2
   cutOffAngle 1.5
   direction 0 -1 0
   intensity 1
   radius 10
   location 2.25 4 2.25
}
Shape {
   appearance Appearance {
      material Material {
         diffuseColor 1 1 1
         ambientIntensity .5
      }
   }
   geometry
      ElevationGrid {
```

```
        height [0 0 0 0 0 0 0 0 0 0
                              0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0
                0 0 0 0 0 0 0 0 0 0]
        xDimension 10
        zDimension 10
        xSpacing .5
        zSpacing .5
      }
   }
}
```

Listado 49: Malla de caras iluminada por el nodo SpotLight

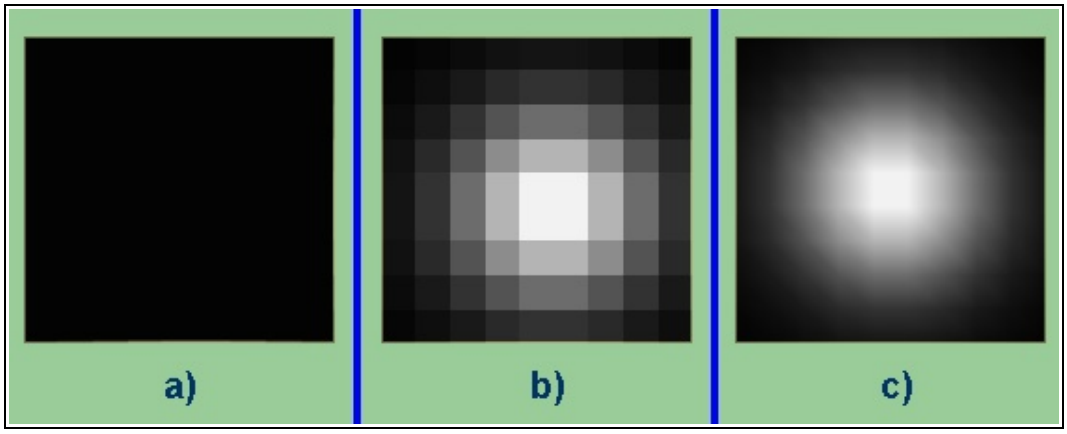


Figura 47: Uso del nodo SpotLight con caras de Sólido

3.5 Sonidos

Un mundo virtual no estaría completo si sólo se pudieran observar modelos espectaculares pero carentes de una ambientación real. Los sonidos vienen a poner este toque de distinción, dotando del ingrediente extra para que la experiencia virtual sea completa.

Para ello se facilitan cualidades de ambientación tridimensional, para que realmente se "perciba" desde donde proviene el sonido, y de atenuación para apreciar una disminución de la intensidad sonora a medida que el *avatar* se aleja de la fuente del sonido.

3.5.1 Sound

Sintaxis:

```
Sound {
      exposedField SFVec3f  direction     0 0 1   # (-∞,∞)
      exposedField SFFloat  intensity     1       # [0,1]
      exposedField SFVec3f  location      0 0 0   # (-∞,∞)
      exposedField SFFloat  maxBack       10      # [0,∞)
      exposedField SFFloat  maxFront      10      # [0,∞)
      exposedField SFFloat  minBack       1       # [0,∞)
      exposedField SFFloat  minFront      1       # [0,∞)
```

```
      exposedField SFFloat  priority      0       # [0,1]
      exposedField SFNode   source        NULL
      field        SFBool   spatialize    TRUE
}
```

Este nodo es el encargado de implementar los sonidos del mundo virtual. Para ello sigue el modelo expuesto en la Figura 48. De esta manera el sonido es emitido desde el punto tridimensional definido por el campo ***location***, en la dirección y sentido indicados por el vector asignado al campo ***direction*** y manteniendo dentro del elipsoide interior el mismo nivel de intensidad sonora definida por el campo ***intensity***. Luego, a medida que nos alejamos el sonido sufre una atenuación lineal hasta los límites del elipsoide exterior a partir de los cuales ya no será audible.

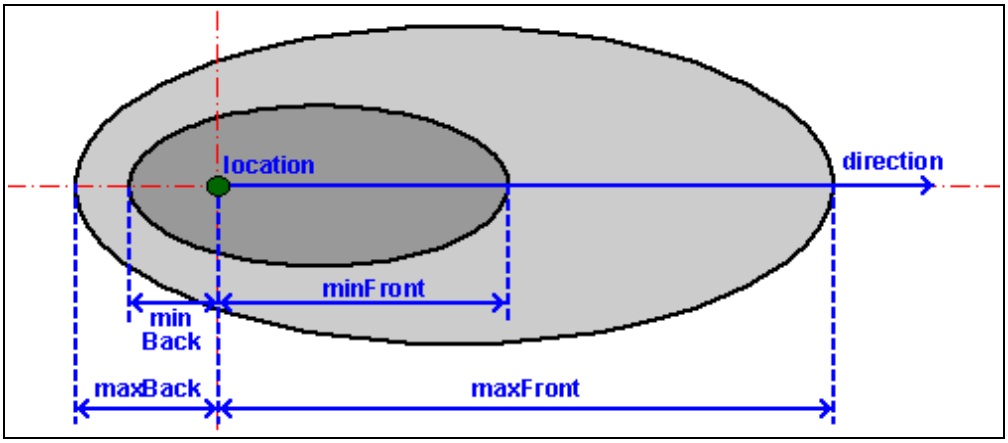


Figura 48: El nodo Sound

Ambos elipsoides vienen definidos por las distancias de los ejes mayor y menor al foco en el cual se ubica la fuente de sonido punto que queda determinado por el campo ***location***. De esta manera, con los campos ***minBack*** y ***minFront***, se define el elipsoide interior y con los campos ***maxBack*** y ***maxFront***, el elipsoide exterior (véase la Figura 48).

El campo ***priority*** cumple la función de auxiliar al *plug-in* en caso de sobrecarga computacional, indicando cuáles sonidos debe reproducir y cuáles debe descartar. Los valores para este campo van desde el 1.0 de máxima prioridad al 0.0 de mínima.

El campo ***spatialize*** controla si el sonido se distribuirá de forma que denote el sitio del que proviene, es decir, si cuando la fuente sonora se encuentra a la izquierda del *avatar*, el sonido tiene una intensidad mayor en el altavoz izquierdo, y viceversa con el derecho. Un valor TRUE en este campo (por defecto) activa esta capacidad, la cual no es un requisito obligatorio para los *plug-ins* según el estándar. Un valor de FALSE la desactiva, siendo igualitaria entonces la distribución de la intensidad sonora entre los altavoces, pero respetándose en todo caso la atenuación progresiva dada por el modelo de los dos elipsoides.

Para finalizar, el campo ***source*** permite especificar la fuente del sonido, que podrá ser un nodo **AudioClip** analizado a continuación, o un nodo **MovieTexture** analizado en el Punto 3.3.1.3. Para un ejemplo de uso véase el Listado 50 en el siguiente punto.

3.5.1.1 AudioClip

Sintaxis:

```
AudioClip {
      exposedField   SFString description      ""
      exposedField   SFBool   loop             FALSE
      exposedField   SFFloat  pitch            1.0         # (0,∞)
      exposedField   SFTime   startTime        0           # (-∞,∞)
      exposedField   SFTime   stopTime         0           # (-∞,∞)
      exposedField   MFString url              []
      eventOut       SFTime   duration_changed
      eventOut       SFBool   isActive
}
```

Este nodo se utiliza para indicar la ubicación del fichero de sonido a ser reproducido por el nodo **Sound**. Para ello se especificará mediante el campo ***url***, la ubicación del fichero fuente que deberá estar en formato *WAV* sin compresión o *MIDI*.

Además se definen ciertas características particulares para el fichero de sonido:

El campo ***description*** que se utiliza para asignar un texto informativo, el cual el *plug-in* no está obligado a mostrar.

El campo ***pitch*** mediante el cual el se podrá cambiar la velocidad por defecto a la cual se reproducirá el sonido. Junto con la velocidad también variará el timbre, subiendo o bajando el mismo acorde al valor de este campo.

Al igual que con el nodo **MovieTexture**, el nodo **AudioClip** cuenta con los campos ***startTime***, ***stopTime*** y ***loop***. Su utilización es similar al caso ya comentado: con ***startTime*** y ***stopTime*** se controlará el comienzo y finalización de la reproducción, mientras que con ***loop*** se indicará si la reproducción finalizará al alcanzarse el fin del fichero de sonido (FALSE), o bien si será cíclica (TRUE). Para más información consulte la explicación del nodo **MovieTexture** en el Punto 3.3.1.3.

En el Listado 50 se presenta un ejemplo de uso de los nodos **Sound** y **AudioClip**.

```
#VRML V2.0 utf8
Sound {
   priority .5
   intensity .8
   source AudioClip {
      stopTime -1
      loop TRUE
      url "sonido.wav"
   }
}
```

Listado 50: Los nodos Sound y AudioClip

3.6 Nodos de Agrupación

En este punto se aborda un conjunto de nodos muy importante en VRML: los nodos de agrupación. Por medio de estos nodos se aplicará el concepto de jerarquía y se aclararán la idea de nodos hijos y de sistemas de coordenadas global y local.

Todos los nodos analizados en esta sección tienen en común los campos ***children***, ***bboxCenter*** y ***bboxSize***. El campo ***children*** definirá a un conjunto de nodos, hasta un máximo de 500 que, como hijos del nodo de agrupación, se verán afectados por el comportamiento del nodo padre.

Los campos ***bboxCenter*** y ***bboxSize*** definen el centro y las dimensiones de una caja invisible que contiene en su interior a la totalidad de los nodos que forman parte del conjunto. La finalidad de esta caja es la de optimizar la presentación del conjunto de nodos. Si no se la especifica, dejando en ambos campos sus valores por defecto, el *plug-in* será el encargado de calcular estos valores lo que para conjuntos numerosos puede consumir bastante recursos computacionales. Si la caja especificada no es capaz de encerrar a la totalidad de los nodos del conjunto, los resultados obtenidos no están definidos por la especificación, siendo dependientes de cada implementación particular.

Los eventos de entrada ***addChildren*** y ***removeChildren*** se utilizarán desde los *scripts* para agregar y eliminar nodos del conjunto de hijos perteneciente a un nodo de agrupación. Véase en el fichero *pizarra.wrl* en el Punto 5.3 para un ejemplo de uso.

3.6.1 Group

Sintaxis:

```
Group {
      eventIn      MFNode  addChildren
      eventIn      MFNode  removeChildren
      exposedField MFNode  children      []
      field        SFVec3f bboxCenter    0 0 0     # (-∞,∞)
      field        SFVec3f bboxSize      -1 -1 -1  # (0,∞) ó -1,-1,-1
}
```

El nodo **Group** cumple la función de agrupar nodos en un mismo nivel de jerarquía. Mediante el campo ***children*** se especificará el conjunto de nodos que estarán en el nivel jerárquico inmediatamente inferior (véase la Figura 49)

En el Listado 51 se observa un ejemplo de uso del nodo **Group** correspondiéndose con el esquema representado en la Figura 49.

```
#VRML V2.0 utf8
Group {
      children [
            Shape {
                  geometry Box { }
            }
            Shape {
                  geometry Cone { }
            }
            Shape {
                  geometry Sphere { }
            }
      ]
}
```

Listado 51: El nodo Group

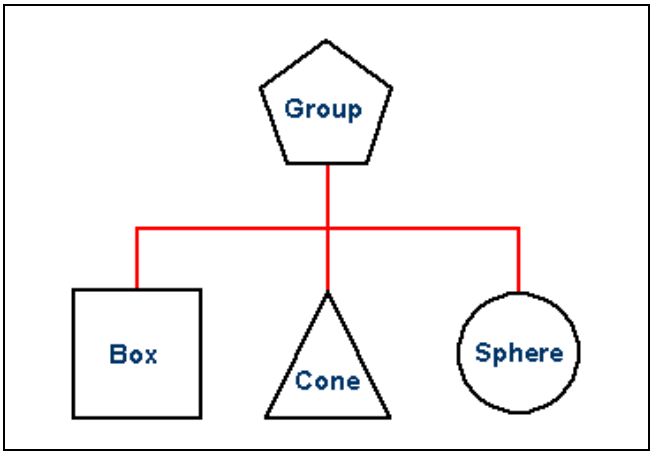


Figura 49: El nodo Group

3.6.2 Transform

Sintaxis:

```
Transform {
      eventIn      MFNode      addChildren
      eventIn      MFNode      removeChildren
      exposedField SFVec3f     center           0 0 0    # (-∞,∞)
      exposedField MFNode      children         []
      exposedField SFRotation  rotation         0 0 1 0  # [-1,1],(- ∞,∞)
      exposedField SFVec3f     scale            1 1 1    # (0, ∞)
      exposedField SFRotation  scaleOrientation 0 0 1 0  # [-1,1],(- ∞, ∞)
      exposedField SFVec3f     translation      0 0 0    # (-∞,∞)
      field        SFVec3f     bboxCenter       0 0 0    # (-∞,∞)
      field        SFVec3f     bboxSize         -1 -1 -1 # (0, ∞) ó -1,-1,-1
}
```

El nodo **Transform** cumple la función de ubicar a sus hijos dentro del mundo virtual, sirviéndose de sus campos de transformación permite trasladar, escalar y/o rotar al conjunto de nodos que se indiquen en su campo ***children***.

El campo ***translation*** permite especificar un conjunto de coordenadas (**x y z**) de traslado relativo a sistema de coordenadas local. Y aquí se hace la salvedad de "local", debido a que se pueden anidar más de un nodo **Transform**, acumulándose las transformaciones a medida que se avanza en profundidad por la jerarquía.

En el Listado 52 se observa el uso del nodo **Transform** aplicando traslaciones a una caja y a una esfera, mientras el cono permanece centrado en el origen de coordenadas. Véase en la Figura 50 los resultados obtenidos.

```
#VRML V2.0 utf8
# CENTRADO EN EL ORIGEN
Shape {
   geometry Cone { }
}
# DESPLAZAMIENTO HACIA ARRIBA (EJE Y)
Transform {
   translation 0 4 0
   children
      Shape {
         geometry Box { }
      }
}
# DESPLAZAMIENTO DIAGONAL
Transform {
   translation 3 2 0
   children
      Shape {
         geometry Sphere { }
      }
}
```

Listado 52: Uso del campo translation del nodo Transform

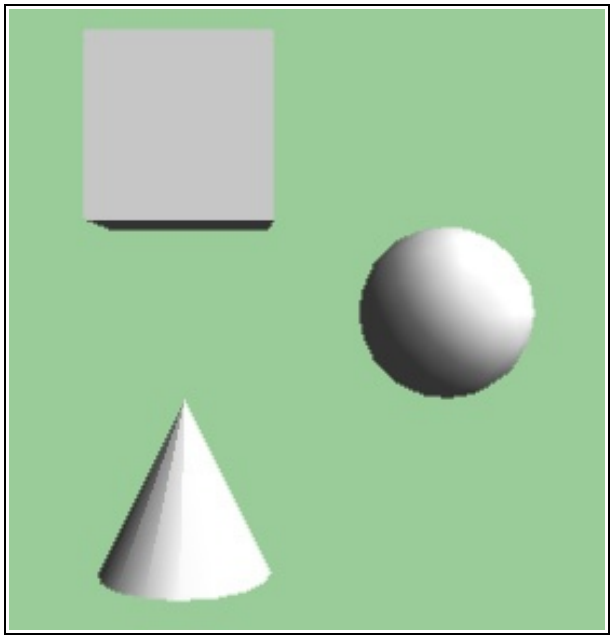


Figura 50: Uso del campo translation del nodo Transform

El campo ***rotation*** especifica la rotación del conjunto de hijos respecto al eje definido por el vector tridimensional, representado por las tres primeras componentes de este campo, y que pasa por el punto indicado en el campo ***center***. Se rotará el número de grados en radianes, indicado como cuarta componente. Para determinar el sentido de giro se sigue la regla de la mano derecha, tal como se explicó en la Figura 1.

En el Listado 53 se presenta el código fuente para realizar tres tipos de rotaciones sobre una caja. Primero se modela una caja sin transformaciones y centrada en el origen de coordenadas, luego a la segunda caja se la rota unos 0.7854 radianes (PI/4) desplazada para evitar superposiciones. Posteriormente la tercera caja se modela aplicándosele la misma rotación pero sobre un eje desplazado. Por último la cuarta caja presenta una rotación sobre un eje oblicuo.

```
#VRML V2.0 utf8
# SIN ROTACION
Shape {
   geometry Box {size 1 2 2}
}
# ROTACION
Transform {
   translation 3 3 0
   rotation 0 0 1 0.7854
   children
      Shape {
         geometry Box {size 1 2 2 }
      }
}
# ROTACION SOBRE CENTRO DESPLAZADO
Transform {
   translation 0 3 0
   center -.5 -1 0
   rotation 0 0 1 0.7854
   children
      Shape {
         geometry Box {size 1 2 2 }
      }
}
# ROTACION SOBRE EJE OBLICULO
Transform {
   translation 3 0 0
   rotation 1 1 1 0.7854
   children
      Shape {
         geometry Box {size 1 2 2 }
      }
}
```

Listado 53: Uso del campo rotation del nodo Transform

En la Figura 51 se pueden observar los resultados producidos por este listado. Nótese que se han agregado externamente los ejes utilizados para las rotaciones para facilitar así la comprensión de la figura.

El campo ***scale*** indicará una deformación escalar de los nodos hijos en las tres dimensiones espaciales. Una componente que valga uno indicará un tamaño sin modificaciones; un valor menor que 1 y mayor que 0, una reducción; y un valor mayor que 1 un aumento de tamaño. Mediante el campo ***scaleOrientation*** se definirá un eje y un ángulo de rotación sobre el cual aplicar la escala.

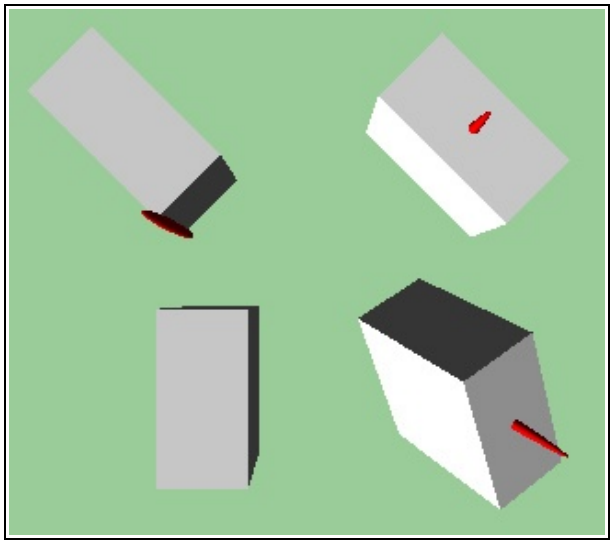


Figura 51: Uso del campo rotation con el nodo Transform

En el Listado 54 se observa un ejemplo de uso del campo ***scale*** del nodo **Transform** aplicado a una esfera. El primer caso, como en el resto de los ejemplos de este nodo, se presenta la primitiva sin transformaciones para ser luego comparada. Posteriormente se modela una esfera arriba de la primera, en la cual se reduce su dimensión **y** a la tercera parte. Por último, a la tercera esfera se le aplica la misma reducción pero en este caso variando el eje escalar.

```
#VRML V2.0 utf8
# SIN ESCALADO
Transform {
   children
      Shape {
         geometry Sphere { }
      }
}
# ESCALADO SOBRE EL EJE Y
Transform {
   translation 0 3 0
   scale 1 0.3333 1
   children
      Shape {
         geometry Sphere { }
      }
}
# ESCALADO DIRIGIDO
Transform {
   translation 3 1.5 0
   scale 1 .3 1
   scaleOrientation 0 .707 .707 1.5707
   children
      Shape {
         geometry Sphere { }
      }
}
```

Listado 54: Uso del campo scale con el nodo Transform

En la Figura 52 se pueden observar las tres esferas y las transformaciones realizadas sobre las mismas.

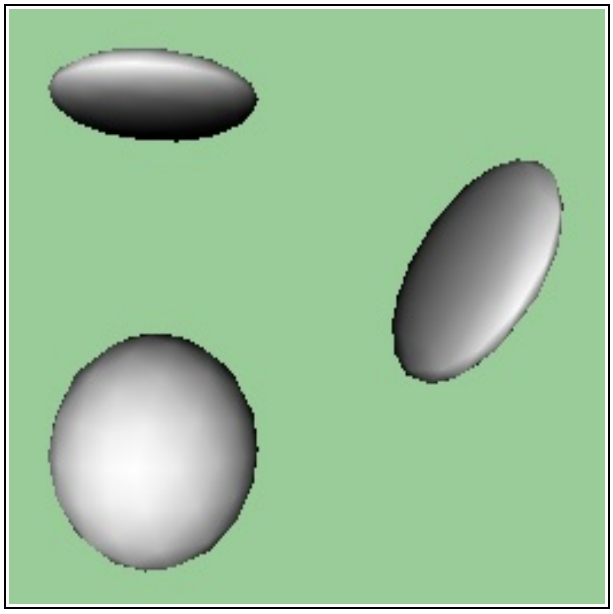


Figura 52: Uso del campo scale con el nodo Transform

Cuando se especifique más de una transformación para un mismo nodo **Transform**, el orden de aplicación será el siguiente: Primero se producirá el escalado del conjunto contenido en el campo ***children*** en el sentido y la orientación indicados para la escala, a continuación se producirá la rotación en su sentido y orientación propios, y por último se efectuará la traslación requerida.

3.6.3 Billboard

Sintaxis:

```
Billboard {
      eventIn      MFNode   addChildren
      eventIn      MFNode   removeChildren
      exposedField SFVec3f  axisOfRotation 0 1 0     # (-∞,∞)
      exposedField MFNode   children       []
      field        SFVec3f  bboxCenter     0 0 0     # (-∞,∞)
      field        SFVec3f  bboxSize       -1 -1 -1  # (0,∞) ó -1,-1,-1
}
```

El nodo **Billboard** es un nodo de agrupación, el cual se encarga de rotar a los componentes de su conjunto hijo de manera que orienten siempre la misma cara hacia el observador.

El campo ***children*** contiene al conjunto de los nodos hijos del nodo **Billboard** que serán los afectados por la transformación que este introduzca.

El campo ***axisOfRotation*** define el eje alrededor del cual se producirá la rotación del conjunto de nodos. Un caso particular es el vector (0, 0, 0) que representará una rotación en todos los sentidos, de manera que el eje **y** del conjunto de nodos hijos, quede siempre paralelo con el eje **y** del *avatar*.

El Listado 55 presenta un ejemplo de uso del nodo **Billboard** en el cual se controla la orientación de un cono rotándolo alrededor del eje **z**.

```
#VRML V2.0 utf8
Billboard {
   axisOfRotation 1 0 0
   children
      Shape {
         appearance Appearance {
            material Material {
               diffuseColor 0 0 1
            }
         }
         geometry Cone { }
      }
}
# CONTINUA EL CÓDIGO CON EL OTRO CONO Y LA CAJA DE LA BASE
```

Listado 55: El nodo Billboard

En la Figura 53 se observa el cono del listado precedente, junto a otro cono y una caja que no pertenecen al conjunto de hijos del nodo **Billboard**. Nótese como el cono azul rota junto con el observador de manera que siempre presenta la misma orientación hacia el *avatar*.

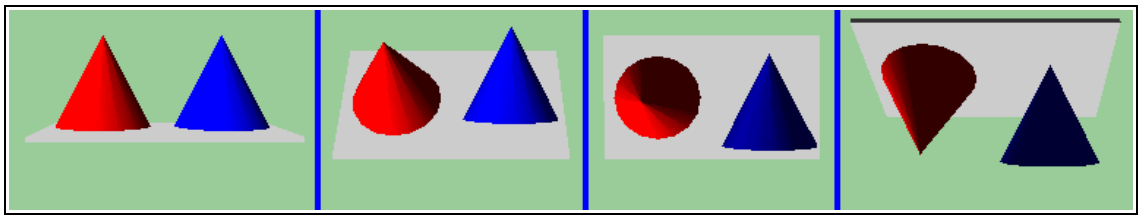


Figura 53: El nodo Billboard

3.6.4 Collision

Sintaxis:

```
Collision {
      eventIn      MFNode   addChildren
      eventIn      MFNode   removeChildren
      exposedField MFNode   children        []
      exposedField SFBool   collide         TRUE
      field        SFVec3f  bboxCenter      0 0 0      # (-∞,∞)
      field        SFVec3f  bboxSize        -1 -1 -1   # (0,∞) ó -1,-1,-1
      field        SFNode   proxy           NULL
      eventOut     SFTime   collideTime
}
```

El nodo **Collision** cumple las funciones de determinar la forma en que se comportan sus nodos hijos respecto a las colisiones con el *avatar*. Por defecto todas las primitivas, salvo **IndexedLineSet**, **PointSet** y **Text**, no pueden ser atravesadas por el *avatar* produciéndose una colisión en el momento en que éste se encuentre a una determinada distancia, véase el nodo **NavigationInfo** en el Punto 3.9.4.

El campo ***children*** define a los hijos de este nodo, los cuales verán modificado su comportamiento frente a las colisiones.

El campo ***collide*** contiene a un valor booleano, indicando si los nodos hijos producirán colisiones cuando el *avatar* intente atravesarlos, al valer TRUE (valor por defecto), o por el contrario podrán ser atravesados, al tomar el valor FALSE.

El campo ***proxy*** admite la inclusión de una primitiva, o un conjunto de primitivas (usando el nodo **Group**), que en lugar de ser renderizadas serán empleadas para que el *avatar* colisione contra ellas. De esta manera se permite definir una geometría de colisión distinta a la del modelo visible.

Véase en el Listado 56 un ejemplo de uso del nodo **Collision** en donde se ha modelado un cubo al cual el *avatar* no puede acercarse, gracias al empleo de una esfera como elemento de colisión, simulando una especie de "campo de fuerza".

```
#VRML V2.0 utf8
Collision {
   proxy
      Shape {
         geometry Sphere {radius 4}
      }
   children
      Shape {
         appearance Appearance {
            material Material {
               diffuseColor 1 0 1
            }
         }
         geometry Box { }
      }
}
```

Listado 56: El nodo Collision

En la Figura 54 se observa el modelo del listado anterior. Nótese que se ha dibujado de forma semitransparente a la esfera que se comporta como elemento de colisión. Esto se ha hecho con fines pedagógicos debido a que en realidad, los nodos utilizados en el campo ***proxy*** no son visibles.

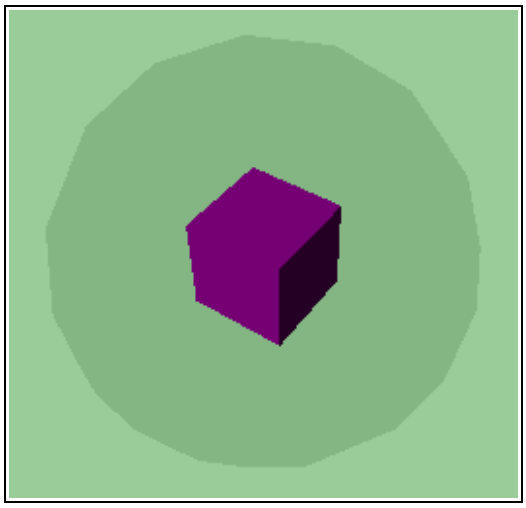


Figura 54: El nodo Collision

3.6.5 Inline

Sintaxis:

```
Inline {
      exposedField MFString url        []
      field        SFVec3f  bboxCenter 0 0 0     # (-∞,∞)
      field        SFVec3f  bboxSize   -1 -1 -1  # (0,∞) ó -1,-1,-1
}
```

El nodo **Inline** permite incluir dentro de un fichero VRML el contenido de otro mundo leído desde un fichero local o desde Internet, empleando su contenido como sus nodos hijos.

El campo ***url*** permite especificar el conjunto de direcciones de Internet (o locales) desde donde leer los nodos hijos. Este fichero debe contener código VRML, de no ser así los resultados obtenidos son imprevisibles.

El momento en que se comienzan a cargar los datos desde el fichero externo queda a cargo de la implementación del *plug-in*, aunque normalmente en mundos de gran complejidad, su carga se postergará hasta que las entidades entren en el campo visual del *avatar*.

En el Listado 57 se presenta un ejemplo de uso del nodo **Inline**, el cual inserta el contenido del fichero *modelo.wrl* desplazado 2 metros en el eje **x**.

```
#VRML V2.0 utf8
Transform {
      translation 2 0 0
      children
            Inline {
                  url "modelo.wrl"
            }
}
```

Listado 57: El nodo Inline

Es común utilizar en el desarrollo de mundos virtuales complejos, nodos **Inline** con fines de reutilización de modelos comunes a varias zonas. Extintores, bancos, macetas, mesas, sillas, computadores, habitaciones completas, son ejemplos de primitivas factibles de reutilizar y las cuales no será necesario volver a definir. Además, su uso permitirá la carga progresiva de las distintas secciones del mundo, mejorando el aspecto general en los primeros instantes de carga del modelo.

3.6.6 InlineLoadControl

Sintaxis:

```
InlineLoadControl {
      exposedField SFBool    load         TRUE
      exposedField MFString  url          []
      field        SFVec3f   bboxCenter   0 0 0       # (-∞,∞)
      field        SFVec3f   bboxSize     -1 -1 -1    # [0,∞) ó (-1,-1,-1)
      eventOut     MFNode    children
```

```
}
```

El nodo **InlineLoadControl** al igual que le nodo **Inline**, lee desde Internet un fichero, indicado en el campo ***url***, en el cual se especifican sus nodos hijos.

Su comportamiento es similar al nodo citado, con la salvedad que el campo ***load*** determinará cuándo debe cargarse el contenido del fichero referenciado. Si su valor TRUE (por defecto) este nodo se comportará al igual que el nodo **Inline**, mientras que si este campo vale FALSE no habrá transferencia alguna. De esta manera, mediante el uso del evento ***set_load*** enviado por un **ProximitySensor** por ejemplo (véase el Punto 3.7.6), se podrá indicar el momento preciso en que deberá cargarse el fichero referenciado.

El Listado 58 presenta un ejemplo de uso del nodo **InlineLoadControl**.

```
#VRML V2.0 utf8
InlineLoadContro {
      url "modelo.wrl"
      load FALSE
}
```

Listado 58: El nodo InlineLoadControl

3.6.7 Switch

Sintaxis:

```
Switch {
      exposedField    MFNode  choice      []
      exposedField    SFInt32 whichChoice -1    # [-1,∞)
}
```

El nodo **Switch** permite seleccionar uno de sus hijos, que en este caso se encuentran indexados, para representarse en el mundo virtual.

El campo ***choice*** contiene una lista con los nodos o grupo de nodos de los cuales, sólo uno será seleccionado para modelarse, o bien, ninguno.

El campo ***whichChoice*** contiene el índice para seleccionar de la lista de nodos del campo ***choice***. Su valor puede ser –1, en cuyo caso no se selecciona ningún hijo, o un número entre 0 (primer hijo) y el número de hijos menos uno. Si el valor del campo ***whichChoice*** es mayor o igual al número de hijos, tampoco se seleccionará ningún nodo.

La verdadera utilidad de este nodo se da mediante el cambio del valor del campo ***choice*** a voluntad haciendo uso de un *script* de código.

Es importante notar que las entidades que no se rendericen por no estar seleccionadas por este nodo continuaran recibiendo y enviando eventos.

En el Listado 59 se presenta un ejemplo de uso del nodo **Switch**. Nótese que al cargarse el fichero sólo se modelará una caja dentro del mundo virtual (índice 0). Obsérvese también que para el nivel 2 se ha definido un grupo (nodo **Group**) para poder agrupar dos nodos dentro de un mismo nivel. Las comas se han escrito con fines aclarativos, separando un nivel de otro, pero como ya se ha comentado en otras ocasiones VRML no las tiene en cuenta pudiéndose obviar su uso.

```
#VRML V2.0 utf8
Switch {
      choice [
            Shape {
                  geometry Box { }
            }
            ,
            Shape {
                  geometry Cone { }
            }
            ,
            Group {
                  children [
                        Shape {
                              geometry Sphere { }
                        }
                        Shape {
                              geometry Cylinder { }
                        }
                  ]
            }
      ]
      whichChoice 0
}
```

Listado 59: El nodo Switch

3.6.8 LOD

Sintaxis:

```
LOD {
      exposedField MFNode  level    []
      field        SFVec3f center   0 0 0    # (-∞,∞)
      field        MFFloat range    []       # (0,∞)
}
```

El nodo **LOD** se utiliza para controlar el grado de detalle con el que se presentan los modelos en el mundo virtual. Su función es similar a la del nodo **Switch** pero en este caso la selección del hijo que será modelado en cada momento se lleva a cabo en función de la distancia a la que se encuentre el *avatar* en ese instante.

El campo ***level*** contendrá la lista de nodos a representar en cada nivel. Al igual que con el campo ***choice*** del nodo **Switch** se podrán utilizar grupos dentro de cada nivel.

El campo ***center*** define un punto en el sistema de coordenadas local, a partir del cual se medirá la distancia hacia el observador para seleccionar el nivel a modelar.

El campo ***range*** contendrá una lista con los valores de los subrangos que se corresponderán con cada nivel de representación. Si se especifican más subrangos que niveles en el campo ***level***, para los intervalos de menor nivel de detalle (más lejanos) se reutilizará el último nivel. Mientras que si se definen más niveles que subrangos, los de menor nivel de detalle serán ignorados. Existe una característica muy potente de VRML que lamentablemente no está implementada en todos los *plug-ins* y que consiste en que, si se especifica un rango vacío, el mismo *plug-in* será el encargado de seleccionar el nivel más apropiado de representación, acorde no solamente a la distancia del *avatar* sino que también a la carga computacional actual del sistema.

Al igual que con el nodo **Switch**, los nodos que no se rendericen al no pertenecer a un nivel activo continúan recibiendo y enviando eventos.

El Listado 60 presenta un ejemplo de utilización del nodo **LOD**. Nótese que las primitivas empleadas son las mismas que en el Listado 59 utilizado como ejemplo para el nodo **Switch**, pero en este caso la conmutación entre niveles vendrá determinada por la distancia entre el *avatar* y el punto (0 0.5 0) indicado en el campo ***center***. Si el *avatar* se encuentra a una distancia menor a 10 metros se modelará una caja, si la distancia está entre 10 y 20 metros se modelará un cono, y por último si la distancia es mayor a 20 metros se modelará una esfera y un cilindro.

```
#VRML V2.0 utf8
LOD {
      level [
            Shape {
                  geometry Box { }
            }
            ,
            Shape {
                  geometry Cone { }
            }
            ,
            Group {
                  children [
                        Shape {
                              geometry Sphere { }
                        }
                        Shape {
                              geometry Cylinder { }
                        }
                  ]
            }
      ]
      range [10,20]
      center 0 0.5 0
}
```

Listado 60: El nodo LOD

3.7 Sensores

Los sensores son nodos capaces de generar eventos de salida, para que puedan ser enrutados hacia uno o más nodos que contengan un receptor de eventos de entrada del mismo tipo. La encargada de definir estos vínculos es la palabra reservada **ROUTE**, que como si fuera un cable establecerá una conexión.

Los campos públicos (*exposedField*) pueden funcionar como generadores de eventos de salida (*eventOut*), como receptores de eventos de entrada (*eventIn*), o como un campo normal de un nodo (*field*). Es más, cuando un campo público recibe un evento de entrada, genera también un evento de salida con el mismo valor y hora de producción que el evento recibido. Para diferenciar la función que desempeña este tipo de campo, se podrá opcionalmente agregar el prefijo ***set_*** cuando sea utilizado como evento de entrada, y el sufijo ***_changed*** cuando lo sea como evento de salida.

Cada evento tiene asociado una hora de producción para permitir el procesamiento ordenado de los mismos, especialmente dentro de un *script* en el que se pueden recibir una gran variedad de eventos en un lapso reducido de tiempo. El valor de la hora de producción también puede ser aprovechado por las funciones del *script* para realizar cálculos internos, o para redirigirlo a eventos de salida (véase el Punto 3.11 en donde se aborda la explicación del nodo **Script**).

Resumiendo, los sensores son los encargados de provocar los comportamientos particulares de cada mundo, permitiendo animaciones e interacciones con el visitante. La recepción de un evento por parte de un nodo o *script* ocasionará un cambio en la escena o la generación y propagación de otros eventos. Por último, en el caso de existir rutas redundantes entre eventos, las mismas serán ignoradas por el *plug-in*.

En general el comportamiento de los nodos frente a los eventos de entrada, será reemplazar dinámicamente el contenido de uno de sus campos por el valor recibido, o la producción de un evento de salida al haber cambiado algún parámetro interno, como la duración de un sonido o la posición de una primitiva.

Precisamente, debido al comportamiento especial de los nodos que producen eventos de salida como los que se enumeran en la Tabla 3 serán tratados expresamente a continuación. El resto, correspondientes a los sensores y a los interpoladores, se tratarán más adelante de forma individual.

Evento de salida	Nodo	Descripción
duration_changed	**AudioClip** **MovieTexture**	Se produce al cambiar la duración del fichero referenciado por el campo ***url*** de estos nodos, normalmente a no estar disponible la primera referencia y pasarse a alguna de las sucesivas. Un valor de –1 indica que aún no se ha terminado con la carga del fichero.
isActive	**AudioClip** **MovieTexture**	TRUE si el fichero referenciado por el campo ***url*** se está reproduciendo y FALSE si ya ha terminado o aún no a comenzado.
collideTime	**Collision**	Produce un evento con la hora en la que se ha producido la colisión entre el *avatar* y alguno de los nodos hijos del nodo.
children	**InlineLoadControl**	Expone la jerarquía cargada desde el fichero especificado en el campo ***url***.

Tabla 3: Eventos de salida

3.7.1 TouchSensor

Sintaxis:

```
TouchSensor {
      exposedField SFBool  enabled   TRUE
      eventOut     SFVec3f hitNormal_changed
      eventOut     SFVec3f hitPoint_changed
      eventOut     SFVec2f hitTexCoord_changed
      eventOut     SFBool  isActive
      eventOut     SFBool  isOver
      eventOut     SFTime  touchTime
}
```

El nodo **TouchSensor** se encarga de detectar cuando el puntero de navegación[28] se encuentra sobre los nodos ubicados en el mismo conjunto que el sensor o en sus descendientes, además se detectará la activación del puntero de navegación como un estado diferente. Ambas situaciones generarán distintas combinaciones de eventos.

El campo ***enabled*** permite indicar la producción (TRUE) o no (FALSE) de los eventos. Nótese que este campo al ser del tipo público, podrá ser modificado externamente a través de un evento de salida proveniente de otro nodo.

El evento de salida ***isOver*** valdrá TRUE, siempre que el puntero de navegación se mueva a una posición sobre alguno de los nodos del conjunto al que pertenece el nodo **TouchSensor** o sobre sus descendientes, y FALSE en otro caso.

[28] El dispositivo de navegación será normalmente el ratón pero también podrá ser otro dispositivo apuntador, incluso unos guantes de realidad virtual. Según el estándar para VRML97 un nodo **TouchSensor** se considerará activo cuando en un dispositivo bidimensional como el ratón, se pulse el botón principal, mientras que para un dispositivo tridimensional se considerará activo cuando se entre en contacto con la superficie sensora propiamente.

El evento de salida ***isActive*** produce una salida con valor TRUE, cuando se active el puntero de navegación estando sobre alguno de los nodos del conjunto (***isOver*** = TRUE) y se mantendrá en este estado a pesar de que se desplace fuera del nodo sensor, siempre y cuando no se desactive el puntero. Este efecto de arrastre es exclusivo del nodo **TouchSensor**.

El evento de salida ***hitPoint_changed*** genera una vector tridimensional con las coordenadas correspondiente al punto sobre el cual se encuentra el puntero de navegación activo, siempre que el mismo pertenezca a alguno de los nodos del conjunto en el que se encuentra el nodo **TouchSensor**.

Lo mismo ocurrirá con los eventos de salida ***hitNormal_changed*** y ***hitTexCoord_changed*** correspondiéndose con el vector normal en el punto y con las coordenadas de la textura respectivamente.

En el Listado 61 se presenta un ejemplo del uso del nodo **TouchSensor.** Nótese que para poder hacer las conecciones por medio de la palabra reservada **ROUTE** ha sido necesario utilizar referencias a los sensores y a los nodos. Para ello se ha utilizado la palabra reservada **DEF**, que será analizada en el Punto 3.10.1. La función de **DEF** será entonces la de proporcionar un alias a un nodo.

```
#VRML V2.0 utf8
Transform {
   translation 0 2 0
   children
      Shape {
         geometry Sphere {radius 0.1}
      }
      DEF TS TouchSensor { }
}
Sound {
   source DEF AC AudioClip {
      stopTime -1
      loop FALSE
      url "click.wav"
   }
}
ROUTE TS.touchTime TO AC.startTime
```

Listado 61: El nodo TouchSensor

En el ejemplo se modela una esfera de radio igual a 0.1 metros y se le asocia un nodo **TouchSensor** llamado “TS” por pertenecer al mismo conjunto hijo del nodo **Transform**. Además se define una fuente sonora mediante el nodo **Sound**, asociando el alias “AC” al nodo **AudioClip**, que es el que contiene la referencia al fichero de sonido.

Mediante la palabra reservada **ROUTE** se conecta el evento de salida ***touchTime*** del alias “TS” con el evento de entrada ***startTime*** del alias “AC” utilizando la palabra reservada **TO** para completar el enlace.

El mundo modelado consistirá en una esfera que al activarse el puntero sobre ella, iniciará la reproducción del sonido almacenado en el fichero *click.wav***.**

3.7.2 PlaneSensor

Sintaxis:

```
PlaneSensor {
      exposedField SFBool  autoOffset          TRUE
      exposedField SFBool  enabled             TRUE
      exposedField SFVec2f maxPosition         -1 -1     # (-∞,∞)
      exposedField SFVec2f minPosition         0 0       # (-∞,∞)
      exposedField SFVec3f offset              0 0 0     # (-∞,∞)
      eventOut     SFBool  isActive
      eventOut     SFVec3f trackPoint_changed
      eventOut     SFVec3f translation_changed
}
```

El nodo **PlaneSensor** permite detectar el comportamiento del puntero de navegación respecto a los nodos pertenecientes al mismo conjunto y a sus hijos. La diferencia principal con el nodo **TouchSensor** se encuentra en que la detección sólo se producirá en el plano **z** = 0 del sistema de coordenadas local.

El campo ***enabled*** habilitará la producción de eventos al valer TRUE, o bien impedirá los mismos si se le asigna un valor FALSE.

El evento de salida ***isActive*** generará un valor TRUE en el momento de activar el puntero de navegación, y un valor FALSE al desactivarlo.

Los eventos de salida ***trackPoint_changed*** y ***translation_changed*** se producen en el momento de activar el puntero de navegación y continúan enviándose hasta que se le desactive. El primero contiene la información respecto al punto sobre el cual se encuentra el puntero, y el segundo, sobre las componentes del vector que tiene como origen el punto en el cual se ha activado el puntero y como extremo el que se encuentra actualmente (siempre que el movimiento se haya realizado sin desactivarlo) lo que se conoce como “arrastre”.

El campo ***offset*** define un punto en el espacio a partir del cual empezar a calcular las componentes del vector de desplazamiento del evento de salida ***translation_changed***, además si el campo ***autoOffset*** vale TRUE (por omisión), el valor del campo ***offset*** se actualizará en cada operación de “arrastre” con el último valor del evento, mientras que si vale FALSE, el valor del campo siempre volverá a su valor inicial.

Los campos ***maxPosition*** y ***minPosition*** definen el rango entre los que variarán los valores emitidos por el evento de salida ***translation_changed***. Nótese la necesidad de incluir estos campos debido a que mientras se mantenga activo el puntero se seguirán emitiendo los eventos ***trackPoint_changed*** y ***translation_changed*** a pesar de abandonar la superficie de las primitivas utilizadas como sensores.

En el Listado 62 se presenta un ejemplo de uso del nodo **PlaneSensor** el cual se encuentra asociado a una caja de dimensiones y aspecto predeterminados. Además en la escena se ha modelado una esfera de

radio igual a 10 centímetros. Mediante el "arrastre" del puntero de navegación sobre la caja se podrá desplazar a voluntad la esfera, enviando los valores producidos por el sensor apodado "PS" como valores de translación del nodo **Transform** apodado "TR".

```
#VRML V2.0 utf8
Transform {
   translation 0 3 0
   children [
      Shape {
         geometry Box { }
      }
      DEF PS PlaneSensor { }
   ]
}
DEF TR Transform {
   children
      Shape {
         geometry Sphere {radius 0.1}
      }
}
ROUTE PS.translation_changed TO TR.set_translation
```

Listado 62: El nodo PlaneSensor

3.7.3 SphereSensor

Sintaxis:

```
SphereSensor {
      exposedField SFBool     autoOffset        TRUE
      exposedField SFBool     enabled           TRUE
      exposedField SFRotation offset            0 1 0 0  # [-1,1],(-∞,∞)
      eventOut     SFBool     isActive
      eventOut     SFRotation rotation_changed
      eventOut     SFVec3f    trackPoint_changed
}
```

El nodo **SphereSensor** detecta la pulsación y "arrastre" del puntero de navegación, a través de una esfera invisible centrada en el origen de coordenadas local; los nodos situados en el mismo conjunto que el sensor determinarán en qué situaciones se producen los eventos que éste genera.

El campo ***enabled*** al valer TRUE habilitará la producción de eventos, o bien desactivará los mismos si toma el valor FALSE.

El evento de salida ***isActive*** producirá un valor TRUE al activar el puntero de navegación sobre los nodos asociados al sensor y un valor FALSE al desactivarlo.

Los eventos de salida ***trackPoint_changed*** y ***rotation_changed*** se producen en el momento de activar el puntero de navegación y se continúan enviando hasta que se lo desactive. El primero contiene la información respecto al punto sobre el cual se encuentra el puntero y el segundo a las componentes del eje y el ángulo de giro que se forma al realizar el movimiento de "arrastre".

El campo ***offset*** define un eje de giro normalizado y un ángulo a partir de los cuales se empezarán a calcular las nuevas rotaciones provistas por el evento de salida ***rotation_changed***, además si el campo ***autoOffset*** vale TRUE (valor por defecto) el campo ***offset*** se actualizará en cada operación de "arrastre" con el último valor del evento, mientras que si vale FALSE su valor siempre volverá al inicial al desactivarse el puntero.

El Listado 63 presenta como ejemplo de uso del nodo **SphereSensor** una esfera modelada con las dimensiones predeterminadas y asociada al sensor esférico apodado "SS"; además se ha modelado una caja de 2 x 0.1 x 0.1 metros como nodo hijo del nodo **Transform** apodado "TR". Mediante la conexión del evento de salida ***rotation_changed*** del nodo "SS" con el campo público ***rotation*** del nodo "TR" (recuérdese que por motivos de claridad se le agregaba el prefijo ***set_*** cuando se comportaba como evento de entrada), se consigue que al "arrastrar" sobre la esfera el puntero, la caja rote sobre su centro siguiendo el movimiento producido por éste sobre el sensor.

```
#VRML V2.0 utf8
Transform {
   translation 0 3 0
   children [
      Shape {
         geometry Sphere { }
      }
      DEF SS SphereSensor {}
   ]
}
DEF TR Transform {
   children
      Shape {
         geometry Box {size 2 0.1 0.1}
      }
}
ROUTE SS.rotation_changed TO TR.set_rotation
```

Listado 63: El nodo SphereSensor

3.7.4 CylinderSensor

Sintaxis:

```
CylinderSensor {
      exposedField SFBool     autoOffset TRUE
      exposedField SFFloat    diskAngle  0.262       # (0,PI/2)
      exposedField SFBool     enabled    TRUE
      exposedField SFFloat    maxAngle   -1          # [-2*PI,2*PI]
      exposedField SFFloat    minAngle   0           # [-2*PI,2*PI]
      exposedField SFFloat    offset     0           # (-∞,∞)
      eventOut     SFBool     isActive
      eventOut     SFRotation rotation_changed
      eventOut     SFVec3f    trackPoint_changed
}
```

El nodo **CylinderSensor** detecta la pulsación y arrastre del puntero sobre un cilindro invisible centrado en el origen de coordenadas local. Los nodos situados en el mismo conjunto que el sensor determinarán en que situaciones se producen los eventos que éste genera.

El campo ***enabled*** al valer TRUE (por defecto) habilitará la producción de eventos, y al valer FALSE desactivará su producción.

El evento de salida ***isActive*** emitirá un valor TRUE al activar el puntero sobre alguno de los nodos asociados al sensor, y un valor FALSE al desactivarlo.

Los eventos de salida ***trackPoint_changed*** y ***rotation_changed*** se producen en el momento de activar el puntero de navegación y se continúan enviando hasta que se lo desactive. El primero contiene la información respecto al punto sobre el cual se encuentra el puntero, y el segundo a las componentes del eje y el ángulo de giro que se forma al realizar el movimiento de "arrastre". La manera de utilizar el sensor vendrá determinada por el ángulo agudo del radio sobre el que se activa el puntero. Si es menor al valor del campo ***diskAngle***, el sensor es interpretado como un cilindro infinito orientado a lo largo del eje **y** local, el cual irá rotando a medida que se desplaza el puntero desde un lado hacia el otro. Por el contrario, si el ángulo es mayor o igual, el sensor será tratado como un cilindro de radio igual a la menor distancia entre el punto de activación del puntero y el eje **y**.

El campo ***offset*** define un valor para el ángulo de giro inicial a partir del cual calcular las nuevas rotaciones correspondientes al evento de salida ***rotation_changed***, además si el campo ***autoOffset*** vale TRUE (valor predeterminado) el campo ***offset*** se actualizará con cada operación de "arrastre" sobre el sensor con el último valor del evento, mientras que si vale FALSE su valor siempre volverá al inicial cuando se desactive el puntero.

Los valores de los campos ***minAngle*** y ***maxAngle*** restringen los valores para el evento ***rotation_changed***. Si ***minAngle*** es mayor que ***maxAngle*** la restricción no será tomada en cuenta.

```
#VRML V2.0 utf8
Transform {
   translation 0 3 0
   children [
      Shape {
         geometry Cylinder {height 0.2}
      }
      DEF CS CylinderSensor {diskAngle 1.57}
   ]
}
DEF TR Transform {
   children
      Shape {
         geometry Box {size 2 0.5 0.5}
      }
}
ROUTE CS.rotation_changed TO TR.set_rotation
```

Listado 64: El nodo CylinderSensor

El Listado 64 presenta un ejemplo de utilización del nodo **CylinderSensor** asociándolo a un disco creado por medio de un cilindro de 0,2 metros de altura. Mediante el sensor se rotará la misma primitiva que en el ejemplo del punto anterior, una caja de 2 x 0.5 x 0.5 metros. Nótese que se ha especificado un

diskAngle de PI/2 radianes consiguiendo así que al "arrastrar" el puntero sobre la cara superior del cilindro se produzca la rotación total de la caja, comportándose el sensor como si se tratase de un botón rotativo.

3.7.5 Anchor

Sintaxis:

```
Anchor {
      eventIn      MFNode   addChildren
      eventIn      MFNode   removeChildren
      exposedField MFNode   children        []
      exposedField SFString description     ""
      exposedField MFString parameter       []
      exposedField MFString url             []
      field        SFVec3f  bboxCenter      0 0 0     # (-∞,∞)
      field        SFVec3f  bboxSize        -1 -1 -1  # (0,∞) ó -1,-1,-1
}
```

El nodo **Anchor** se utiliza para enlazar un conjunto de primitivas con otro mundo virtual o una página web. Este nodo es además de un sensor de toque, un nodo de agrupación.

El campo ***children*** especifica el conjunto de nodos que serán los encargados al activarse, de producir el cambio de escena.

El campo ***url*** contendrá el enlace (o conjunto de enlaces), al que se accederá por medio de este nodo. La escena a cargar puede ser otro fichero *.WRL* que reemplazará al mundo virtual actual, o bien un enlace a una cámara del mismo mundo (véase el análisis sobre las cámaras en la descripción del nodo **Viewpoint** en el Punto 3.9.3). También podrán especificarse enlaces a otro tipo de ficheros, como documentos HTML, para los cuales el *plug-in* será el encargado de invocar al visualizador apropiado.

El campo ***parameter*** contiene los parámetros extra del enlace, como puede ser el marco de destino. (véase el Listado 65 para un ejemplo de uso de este campo)

```
#VRML V2.0 utf8
# ENLACE A UNA PAGINA WEB EN UN MARCO NUEVO
Anchor {
   url "http://www.informatica.uma.es"
   parameter "target=_blank"
   description "E.T.S.I.I."
   children
      Shape {
         geometry Box { }
      }
}
# ENLACE A UNA VISTA DEL MISMO MUNDO VIRTUAL
Anchor {
   url "#VistaSuperior"
   description "Cambio de Vista"
   children
      Shape {
         geometry Sphere { }
      }
}
```

Listado 65: El nodo Anchor

El campo ***description*** de utiliza para agregar un texto descriptivo del enlace, que luego podrá (aunque no obligatoriamente) ser presentado por el *plug-in* para facilitar las tareas de navegación.

3.7.6 ProximitySensor

Sintaxis:

```
ProximitySensor {
      exposedField SFVec3f    center      0 0 0    # (-∞,∞)
      exposedField SFVec3f    size        0 0 0    # [0,∞)
      exposedField SFBool     enabled     TRUE
      eventOut     SFBool     isActive
      eventOut     SFVec3f    position_changed
      eventOut     SFRotation orientation_changed
      eventOut     SFTime     enterTime
      eventOut     SFTime     exitTime
}
```

El nodo **ProximitySensor** se utiliza para detectar la presencia del *avatar* dentro de una caja invisible definida por medio de los campos ***center*** y ***size***, determinando el primero el punto central de la caja en el origen de coordenadas local y el segundo las dimensiones de la misma.

El campo ***enabled*** define si el sensor enviará eventos (TRUE), o si permanecerá inactivo (FALSE).

Los eventos de salida ***isActive***, ***position_changed*** y ***orientation_changed*** se producirán cuando el *avatar* entre en los dominios del sensor, se desplace dentro de él o varíe su orientación, obteniéndose así la totalidad de la información sobre el comportamiento del visitante dentro del ámbito del sensor.

Los eventos de salida ***enterTime*** y ***exitTime*** completan la información disponible proporcionando los eventos correspondientes a la hora de entrada a la región y a la de salida de la misma por parte del *avatar*.

En el Listado 66 se observa el código fuente de un ejemplo de uso del nodo **ProximitySensor** en el que se lo ha utilizado para que, cuando sea activado por el *avatar*, encienda una luz iluminando la escena que en este caso es una caja verde que se extiende por debajo del sensor, haciendo de suelo. Nótese que en el ejemplo la luz comienza desactivada (***on*** FALSE).

```
#VRML V2.0 utf8
DEF PS ProximitySensor {
      center 0 1 0
      size 3 2 3
}
Shape {
   appearance Appearance {material Material {diffuseColor 0 .4 0 }}
   geometry Box {size 10 0.1 10 }
}
DEF DL DirectionalLight {
   direction 0 -1 0
   on FALSE
}
ROUTE PS.isActive TO DL.on
```

Listado 66: El nodo ProximitySensor

3.7.7 VisibilitySensor

Sintaxis:

```
VisibilitySensor {
      exposedField SFVec3f center   0 0 0      # (-∞,∞)
      exposedField SFBool  enabled  TRUE
      exposedField SFVec3f size     0 0 0      # [0,∞)
      eventOut     SFTime  enterTime
      eventOut     SFTime  exitTime
      eventOut     SFBool  isActive
}
```

El nodo **VisibilitySensor** se encarga de producir eventos cuando la caja invisible definida por sus campos ***center*** y ***size***, entra en el campo visual del *avatar*.

El campo ***enabled*** controla la producción de eventos por parte del sensor; si toma el valor TRUE éste estará habilitado y viceversa si el valor asignado es FALSE.

El evento de salida ***isActive*** valdrá TRUE mientras la caja asociada al sensor permanezca en el campo visual, y FALSE cuando no sea visible. Asimismo los eventos de salida ***enterTime*** y ***exitTime*** se producirán en el instante de activación y desactivación del sensor, propagando el valor de las respectivas horas de estos acontecimientos.

En el Listado 67 se presenta el código fuente de un ejemplo de utilización del nodo **VisibilitySensor**. Nótese que se ha dibujado una caja (nodo **Box**) de iguales dimensiones que la definida en el sensor, para comprobar cuando se encuentra el mismo dentro del campo visual. Las horas de los eventos ***enterTime*** y ***exitTime*** serán conducidas a los campos ***startTime*** y ***stopTime*** del nodo **AudioClip**. De esta manera se consigue que mientras se encuentre (y sólo en ese caso) la caja en el campo visual del *avatar*, se escuche el sonido provisto por el fichero *click.wav*.

```
#VRML V2.0 utf8
Group {
   children [
      DEF VS VisibilitySensor {size 1 1 1}
      Shape {
         geometry Box {size 1 1 1}
      }
   ]
}
Sound {
   source DEF AC AudioClip {
      loop TRUE
      url "click.wav"
   }
}
ROUTE VS.enterTime TO AC.startTime
ROUTE VS.exitTime TO AC.stopTime
```

Listado 67: El nodo VisibilitySensor

3.7.8 TimeSensor

Sintaxis:

```
TimeSensor {
      exposedField SFTime   cycleInterval 1       # (0,∞)
      exposedField SFBool   enabled       TRUE
      exposedField SFBool   loop          FALSE
      exposedField SFTime   startTime     0       # (-∞,∞)
      exposedField SFTime   stopTime      0       # (-∞,∞)
      eventOut     SFTime   cycleTime
      eventOut     SFFloat  fraction_changed      # [0, 1]
      eventOut     SFBool   isActive
      eventOut     SFTime   time
}
```

El nodo **TimeSensor** es el encargado de controlar las rampas de tiempo. Para ello generará una variación continua entre 0 y 1 del valor disponible en el evento de salida ***fraction_changed*** durante el tiempo (en segundos) determinado por el campo ***cycleInterval***.

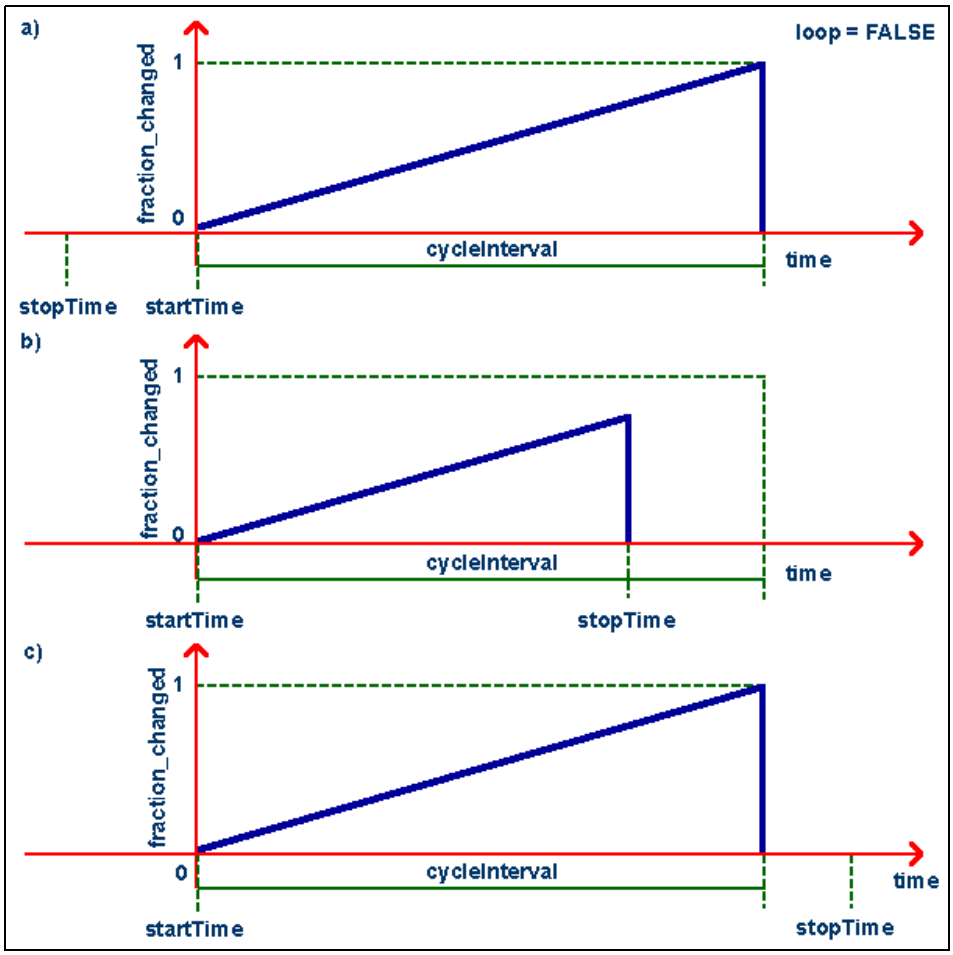


Figura 55: El nodo TimeSensor (loop = FALSE)

Los campos ***startTime*** y ***stopTime*** determinan la hora de comienzo y de finalización de la producción de eventos. De esta manera si ***stopTime*** es menor que ***startTime*** se cumplirá un ciclo en el tiempo determinado por el campo ***cycleInterval*** (véase la Figura 55 **a)**). Mientras que si ***stopTime*** es mayor que

startTime pero menor que ***startTime*** + ***cycleInterval***, la rampa de tiempo no completará su ciclo (véase la Figura 55 **b)**). Por último si ***stopTime*** es mayor que ***startTime*** + ***cycleInterval***, la rampa de tiempo sí completará su ciclo (véase la Figura 55 **c)**).

Dependiendo del campo ***loop***, se reiniciará el ciclo hasta que se cumpla el valor fijado por ***stopTime*** (Figura 56 **b)**), o bien seguirá ininterrumpidamente si el valor de ***stopTime*** es menor que ***startTime*** (Figura 56 **a)**).

El campo ***enabled*** controlará cuándo el sensor emitirá eventos (al valer TRUE) y cuando permanecerá inactivo (al valer FALSE).

El evento de salida ***isActive*** valdrá TRUE mientras el sensor esté generando la rampa de tiempo, pasando a valer FALSE al detenerse.

El evento de salida ***cycleTime*** se producirá cada vez que comience un nuevo ciclo de la rampa de tiempo, incluido el comienzo determinado por ***startTime***. Este evento es útil para sincronizar a otros nodos activos al contener la información de la hora de inicio del ciclo del sensor.

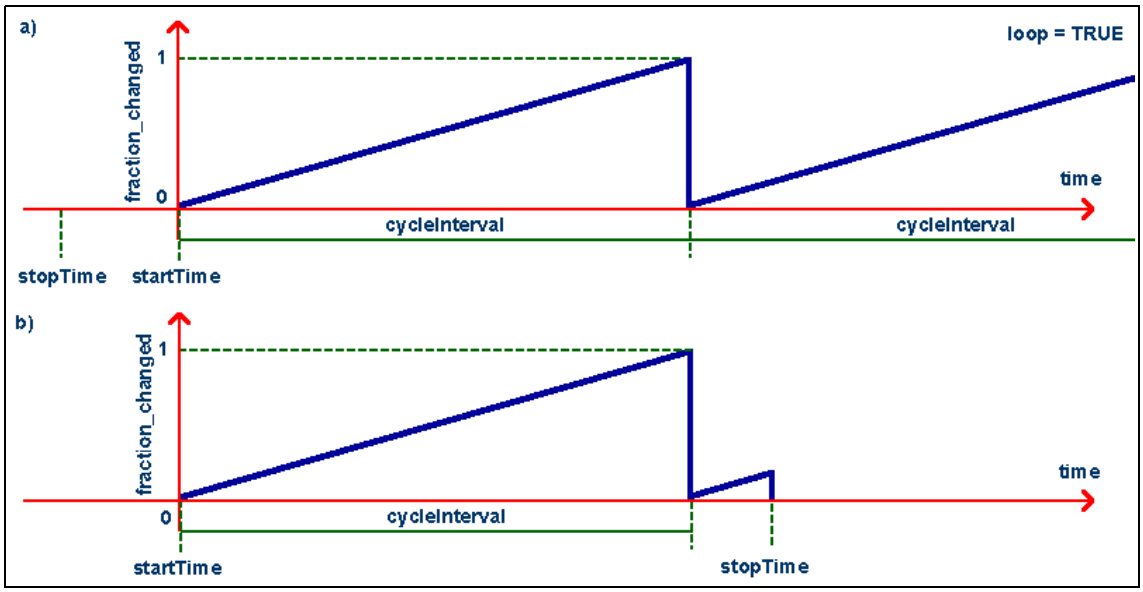


Figura 56: El nodo TimeSensor (loop = TRUE)

El evento de salida ***fraction_changed*** variará su valor entre 0 y 1 a medida que se avanza en el ciclo del sensor, cuya duración viene determinada por el campo ***cycleInterval***.

Por último el evento de salida ***time*** contendrá el valor del tiempo absoluto para cada valor generado por el sensor.

El ejemplo de uso del nodo **TimeSensor** se presenta en el Listado 68 en donde se ha modelado una esfera amarilla, la que será iluminada progresivamente por la luz direccional “DL”, la cual aumenta su intensidad recibiendo los valores de la rampa de tiempo provista por el nodo **TimeSensor** llamado “TS”.

Nótese que el ciclo será continuo al valer TRUE el campo ***loop*** y al ser el valor de ***stopTime*** menor que el de ***starTime***. En el siguiente punto se utilizará a este sensor de forma exhaustiva junto con los interpoladores, para generar los valores que luego serán convertidos a las correspondientes magnitudes.

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {material Material {diffuseColor 1 1 0}}
   geometry Sphere { }
}
DEF DL DirectionalLight {
   intensity 0
   ambientIntensity 0
   direction -1 -1 -1
   on TRUE
}
DEF TS TimeSensor {
   cycleInterval 5
   startTime 0
   stopTime -1
   loop TRUE
}
ROUTE TS.fraction_changed TO DL.intensity
```

Listado 68: El nodo TimeSensor

3.8 Interpoladores

Los interpoladores son nodos pensados especialmente para crear animaciones y dotar de vida al mundo virtual. Su función es la de convertir un valor, que llamaremos clave y cuyos valores normalmente los provee un nodo **TimeSensor** en un desplazamiento, una rotación, un color, etc.

Todos los interpoladores constarán de dos campos públicos: el campo ***key*** en el cual se dispondrán las claves a interpolar, y el campo ***keyValue*** en donde se definirán los valores que se correspondan con cada una de las claves del campo ***key***. La naturaleza del campo ***keyValue*** variará según el tipo de interpolador que se trate.

La lectura de las nuevas claves se produce a través del evento de entrada ***set_fraction*** y la salida interpolada es recogida a través del evento de salida ***value_changed***.

3.8.1 PositionInterpolator

Sintaxis:

```
PositionInterpolator {
      eventIn      SFFloat set_fraction        # (-∞,∞)
      exposedField MFFloat key           []    # (-∞,∞)
      exposedField MFVec3f keyValue      []    # (-∞,∞)
      eventOut     SFVec3f value_changed
}
```

El nodo **PositionInterpolator** convierte los valores recibidos en vectores de tres dimensiones (nótese que también pueden interpretarse como coordenadas).

El Listado 69 presenta un ejemplo de uso del nodo **PostionInterpolator** con el cual se desplaza una esfera haciendo que rebote entre los puntos $x = -2$ y $x = 2$. Nótese que con el nodo **TimeSensor**, alias "TS", se generan los valores entre 0 y 1 continuamente en un ciclo de 5 segundos de duración. Estos valores se envían como claves al nodo **PositionInterpolator**, alias "PI", el cual convierte el rango entre 0 y 0.5 en desplazamientos entre los puntos (–2 0 0) y (2 0 0), y el rango entre 0.5 y 1 en desplazamientos entre (2 0 0) y (-2 0 0). Estos desplazamientos serán encaminados al nodo **Transform**, alias "TR", que plasmará el movimiento de ida y vuelta de la esfera dentro del mundo virtual.

```
#VRML V2.0 utf8
DEF TR Transform {
   children    Shape {geometry Sphere { }}
}
DEF TS TimeSensor {
   cycleInterval 5
   startTime 0
   stopTime -1
   loop TRUE
}
DEF PI PositionInterpolator {
   key      [0,      0.5,     1]
   keyValue [-2 0 0, 2 0 0,   -2 0 0]
}
ROUTE TS.fraction_changed TO PI.set_fraction
ROUTE PI.value_changed TO TR.set_translation
```

Listado 69: El nodo PositionInterpolator

3.8.2 OrientationInterpolator

Sintaxis:

```
OrientationInterpolator {
      eventIn      SFFloat    set_fraction      # (-∞,∞)
      exposedField MFFloat    key           []  # (-∞,∞)
      exposedField MFRotation keyValue      []  # [-1,1],(-∞,∞)
      eventOut     SFRotation value_changed
}
```

El nodo **OrientationInterpolator** se utiliza para convertir cada valor de clave recibida en un eje y un ángulo de rotación. Esta rotación tendrá lugar en el sentido que represente el camino más corto entre dos valores consecutivos, mientras que si ambas alternativas son equidistantes, es decir, si al valor (0 1 0 0) le sigue el valor (0 1 0 3.1416) por ejemplo, la dirección de rotación elegida no está definida en la especificación dependiendo la decisión de cada implementación particular.

El Listado 70 presenta el código fuente de un ejemplo de utilización del nodo **OrientationInterpolator**, asignando a cada clave recibida desde el nodo **TimeSensor** un eje de rotación y un ángulo de giro. Luego, esta rotación será enviada al nodo **Transform** que realizará la correspondiente transformación sobre sus hijos (una caja en este caso). Nótese que se han tenido que definir más claves que las que a priori eran necesarias debido a la necesidad de evitar ambigüedades con

los ángulos del interpolador, especificando siempre el camino más corto entre dos de ellos tal como se ha comentado en el párrafo anterior.

```
#VRML V2.0 utf8
DEF TR Transform {
   children Shape {geometry Box {size 2 .1 .1}}
}
DEF TS TimeSensor {
   cycleInterval 5
   startTime 0
   stopTime -1
   loop TRUE
}
DEF OI OrientationInterpolator {
   key      [0, 0.25, 0.5, 0.75, 1]
   keyValue [0 1 0 0, 0 1 0 1.5708, 0 1 0 3.1416,
             0 1 0 -1.5708, 0 1 0 0]
}
ROUTE TS.fraction_changed TO OI.set_fraction
ROUTE OI.value_changed TO TR.set_rotation
```

Listado 70: El nodo OrientationInterpolator

3.8.3 ScalarInterpolator

Sintaxis:

```
ScalarInterpolator {
      eventIn      SFFloat set_fraction         # (-∞,∞)
      exposedField MFFloat key           []     # (-∞,∞)
      exposedField MFFloat keyValue      []     # (-∞,∞)
      eventOut     SFFloat value_changed
}
```

El nodo **ScalarInterpolator** hace corresponder con cada clave del campo ***key*** un valor del campo ***keyValue***. De esta manera a un rango de valores entre 0 y 1 le corresponderá un rango de valores distinto, pudiéndose especificar una evolución no lineal.

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {material Material {diffuseColor 1 1 0}}
   geometry Sphere { }
}
DEF DL DirectionalLight {
   intensity 0
   ambientIntensity 0
   direction -1 -1 -1
   on TRUE
}
DEF TS TimeSensor {
   cycleInterval 5
   startTime 0
   stopTime -1
   loop TRUE
}
DEF SI ScalarInterpolator {
   key      [0,   0.25, 0.5,  0.75, 1]
   keyValue [1,   0.8,  0.7,  0.6,  0]
}
ROUTE TS.fraction_changed TO SI.set_fraction
ROUTE SI.value_changed TO DL.set_intensity
```

Listado 71: El nodo ScalarInterpolator

En el Listado 71 se observa el código fuente de un ejemplo de uso del nodo **ScalarInterpolator**. Nótese que el ejemplo guarda cierta similitud con el presentado en el Listado 68 como ejemplo de uso del nodo **TimeSensor**, pero en este caso se utiliza al interpolador para convertir los valores. De esta forma en vez de ir aumentando la intensidad de la luz direccional definida por el nodo **DirectionalLight**, esta va apagándose progresivamente, siguiendo una evolución no lineal como se desprende de los valores asociados a las claves del interpolador.

3.8.4 CoordinateInterpolator

Sintaxis:

```
CoordinateInterpolator {
      eventIn      SFFloat set_fraction        # (-∞,∞)
      exposedField MFFloat key           []    # (-∞,∞)
      exposedField MFVec3f keyValue      []    # (-∞,∞)
      eventOut     MFVec3f value_changed
}
```

El nodo **CoordinateInterpolator** produce una variación de un conjunto de coordenadas asociado a cada clave, siguiendo la evolución de los valores recibidos para el campo ***key***.

```
#VRML V2.0 utf8
Shape {
   geometry IndexedLineSet {
      color Color {color [1 1 0, 0 1 1]}
      coord DEF CO Coordinate {point [-1 0 0, 1 1 0]}
      coordIndex [0,1,-1]
   }
}
DEF TS TimeSensor {
   cycleInterval 5
   startTime 0
   stopTime -1
   loop TRUE
}
DEF CI CoordinateInterpolator {
   key      [0, 0.5, 1]
   keyValue [-1 -1 0, 1 1 0, -1 1 0, 1 -1 0, -1 -1 0, 1 1 0]
}
ROUTE TS.fraction_changed TO CI.set_fraction
ROUTE CI.value_changed TO CO.set_point
```

Listado 72: El nodo CoordinateInterpolator

En el Listado 72 se presenta un ejemplo de uso del nodo **CoordinateInterpolator** en el que se varían las coordenadas de dos puntos que conforman un segmento. Nótese que en este caso a cada valor de la clave, le corresponden dos valores del campo ***keyValue***. El número de valores por cada clave vendrá determinado por la cantidad de puntos necesarios para el nodo destino de la interpolación (el apodado "CO" en el ejemplo, sólo requiere dos puntos).

3.8.5 ColorInterpolator

Sintaxis:

```
ColorInterpolator {
      eventIn      SFFloat set_fraction        # (-∞,∞)
      exposedField MFFloat key           []    # (-∞,∞)
      exposedField MFColor keyValue      []    # [0,1]
      eventOut     SFColor value_changed
}
```

El nodo **ColorInterpolator** produce valores de componentes RGB por cada clave asociada a su campo ***key***. El comportamiento no está definido para una interpolación entre dos valores que presenten tonalidades opuestas (componente H del modelo HSB[29]), al igual que ocurría con el nodo **OrientationInterpolator**.

En el ejemplo de código para el nodo **ColorInterpolator** (véase Listado 73), se presenta un cubo modelado por el nodo **Box**, el cual irá variando su color a través de las transiciones definidas para clave del interpolador.

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {material DEF MA Material { }}
   geometry Box { }
}
DEF TS TimeSensor {
   cycleInterval 10
   startTime 0
   stopTime -1
   loop TRUE
}
DEF CI ColorInterpolator {
   key      [0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.750, 0.875, 1]
   keyValue [0 0 0, 1 0 0, 1 1 0, 0 1 0, 0 1 1, 0 0 1, 0 1 1, 1 1 1, 0 0 0]
}
ROUTE TS.fraction_changed TO CI.set_fraction
ROUTE CI.value_changed TO MA.set_diffuseColor
```

Listado 73: El nodo ColorInterpolator

3.8.6 NormalInterpolator

Sintaxis:

```
NormalInterpolator {
      eventIn      SFFloat set_fraction        # (-∞,∞)
      exposedField MFFloat key           []    # (-∞,∞)
      exposedField MFVec3f keyValue      []    # (-∞,∞)
      eventOut     MFVec3f value_changed
}
```

El nodo **NormalInterpolator** es el último interpolador que queda por analizar, igual que los otros variará su magnitud acorde al valor de la clave recibida. En este caso se trata de los vectores para las normales de los nodos **IndexedFaceSet** y **ElevationGrid** los que se obtendrán en el evento de salida ***value_changed***.

[29] Modelo distinto al RGB que define a un color basándose en la percepción del ojo humano. Las tres componentes en este caso son: H (*Hue*) que es la tonalidad reflejada por un objeto, S (*Saturation*) también llamado *chroma* que es la pureza del color y B (*Brightness*) que se corresponde con el nivel de brillo del color.

La restricción para dos valores adyacentes en este caso, vendrá dada cuando dos vectores (que deberán estar normalizados) tengan la misma dirección, y sentido opuesto.

En el Listado 74 se presenta el código fuente de un ejemplo de uso del nodo **NormalInterpolator** el cual aplica distintos valores de normales a la figura modelada con el nodo **IndexedFaceSet**.

```
#VRML V2.0 utf8
Shape {
   appearance Appearance {material Material {diffuseColor 1 1 1}}
   geometry IndexedFaceSet {
      coord Coordinate {point [1 0 0, 1 1 0, -1 1 0, -1 0 0
                               1 0 -1, 1 1 -1]
      }
      coordIndex [0,1,2,3,-1
                  0,4,5,1,-1]
      normal DEF NO Normal { }
      normalPerVertex FALSE
   }
}
DEF TS TimeSensor {
   cycleInterval 5
   startTime 0
   stopTime -1
   loop TRUE
}
DEF NI NormalInterpolator {
   key [0, 0.25, 0.5, 0.750, 1]
   keyValue [0 1 0, 0 1 0, 0 0 1, 1 0 0, 0 1 0, 0 1 0, 1 0 0, 0 0 1, 0 1 0, 0 1 0]
}
ROUTE TS.fraction_changed TO NI.set_fraction
ROUTE NI.value_changed TO NO.set_vector
```

Listado 74: El nodo NormalInterpolator

3.9 Nodos de Entorno

Este grupo de nodos tiene la función de modificar el entorno del usuario, con excepción del nodo **WorldInfo** que se encarga de proporcionar información sobre el mundo virtual al que pertenece. Para ello utilizan una pila (*stack*) individual para cada tipo de nodo. De este modo habrá una pila de nodos **Background,** una de nodos **Fog**, etc.

3.9.1 Background

Sintaxis:

```
Background {
      eventIn      SFBool   set_bind
      exposedField MFFloat  groundAngle  []         # [0,PI/2]
      exposedField MFColor  groundColor  []         # [0,1]
      exposedField MFString backUrl      []
      exposedField MFString bottomUrl    []
      exposedField MFString frontUrl     []
      exposedField MFString leftUrl      []
      exposedField MFString rightUrl     []
      exposedField MFString topUrl       []
      exposedField MFFloat  skyAngle     []         # [0,PI]
      exposedField MFColor  skyColor     0 0 0      # [0,1]
      eventOut     SFBool   isBound
}
```

El nodo **Background** permite definir el fondo de la escena por medio de una esfera conteniendo una gradación de colores o bien mediante seis imágenes, una para cada cara interna del cubo que envuelve al mundo virtual, pudiéndose combinar ambas técnicas.

Los campos ***groundAngle*** y ***groundColor*** definen la extensión de la semiesfera que se corresponde con el suelo (ángulos hasta PI/2), y el gradiente de color que se utilizará para su representación. De esta manera el primer color de la lista correspondiente al campo ***groundColor*** se asignará al ángulo 0 de la esfera (siempre debajo de la línea de visión del *avatar*), el resto de los colores se corresponderán con los intervalos definidos por el campo ***groundAngle***.

Los campos ***skyAngle*** y ***skyColor*** se encargan de los colores para el cielo. Su comportamiento es exactamente igual que el suelo, con la salvedad de que ahora el ángulo 0 está sobre el horizonte del *avatar* y que el valor máximo que se podrá especificar es de PI, por lo que en este caso se trata de una esfera completa.

En el Listado 75 se observa un ejemplo de uso del nodo **Background** en el que se define una semiesfera para el suelo entre 0 y 1 radián, contenida dentro de la esfera para el cielo. Nótese que para el suelo se utilizan tres colores, mientras que para el cielo son cuatro los colores empleados para el degradado.

```
#VRML V2.0 utf8
Background {
   groundAngle [0.7, 1]
   groundColor [0.3 0.3 0.3, 0 0.5 0, 0 0.8 0]
   skyAngle    [0.75, 1.5, 2]
   skyColor    [1 1 1, 0 0.6 0.6, 0 0.6 0.8, 0 0 1]
}
```

Listado 75: El nodo Background

Los resultados obtenidos se observan en la Figura 57, la cual se compuso mediante distintas capturas realizadas variando el ángulo con le que se observaba al mundo virtual. De esta manera la parte superior de la figura se corresponde con lo observado al mirar hacia lo más alto del cielo, mientras que la parte inferior se obtuvo al mirar hacia lo más bajo del suelo.

Los campos ***topUrl***, ***bottomUrl***, ***frontUrl***, ***backUrl***, ***leftUrl*** y ***rightUrl*** se corresponden con las imágenes que se podrán ubicar arriba, abajo, delante, atrás, izquierda y derecha del *avatar* respectivamente. Tomando como referencia la posición inicial por defecto (mirando hacia el sentido decreciente del eje **z**), véase el nodo **Viewpoint** en el Punto 3.9.3 para más información. Estas imágenes podrán contener transparencias permitiendo así que se vea el fondo definido por los campos ***groundColor*** y ***skyColor***.

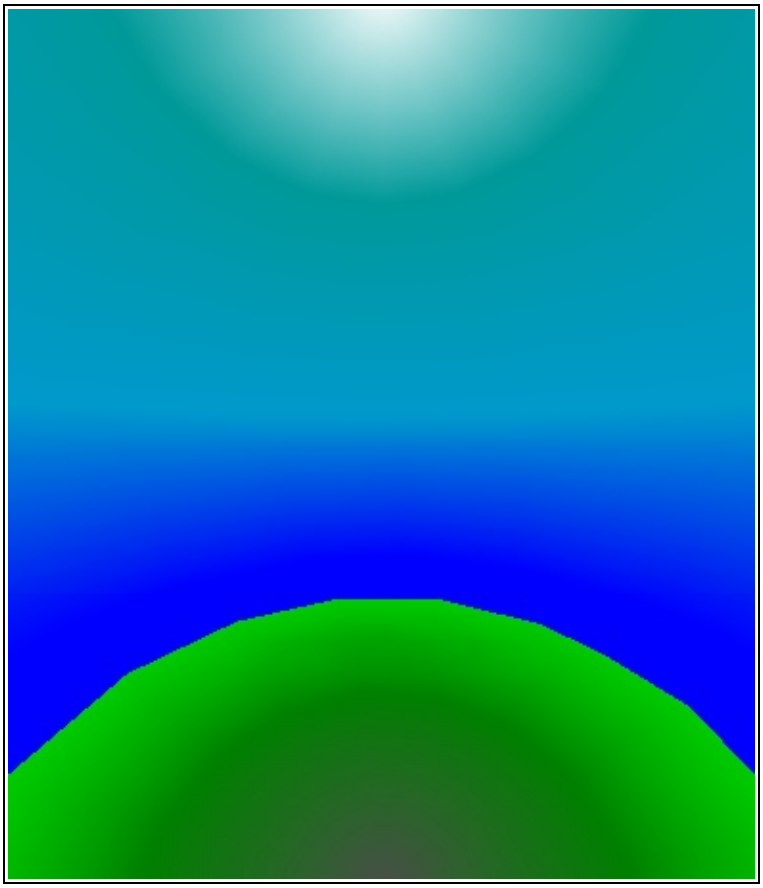


Figura 57: El nodo Background

Como se comentó anteriormente, existe una pila de nodos **Background** de manera que si en un mundo se define más de uno de éstos nodos, estará activo inicialmente el primero que se haya encontrado dentro de la jerarquía del fichero colocándose en la cima de la pila. Luego, por medio del evento de entrada ***set_bind*** se podrá activar cualquiera de los restantes, pasando el mismo a ocupar la cima al recibir un valor TRUE para este campo y volviendo a su posición inicial cuando reciba un valor FALSE.

El evento de salida ***isBound*** producirá un valor TRUE cuando un nodo pase a ocupar la cima de la pila, o cuando ascienda a la misma al ser abandonada por el nodo precedente para el cual el evento de salida ***isBound*** generará un valor FALSE.

El fondo del mundo virtual definido por el nodo **Background** no gira ni se desplaza respecto al *avatar* y su ubicación es infinitamente lejana para que nunca pueda ser alcanzada.

3.9.2 Fog

Sintaxis:

```
Fog {
      exposedField SFColor  color           1 1 1      # [0,1]
      exposedField SFString fogType         "LINEAR"
      exposedField SFFloat  visibilityRange 0          # [0,∞)
      eventIn      SFBool   set_bind
      eventOut     SFBool   isBound
}
```

El nodo **Fog** es el encargado de modelar el efecto de niebla dentro del mundo virtual. Dispone de dos modelos para la misma: “LINEAR” que representa una disminución lineal de la visibilidad con la distancia y “EXPONENTIAL” que genera un efecto más realista pero que no está soportado por todos los *plug-ins*. Este nodo también tiene asociada una pila comportándose de manera similar al nodo **Background**. Téngase en cuenta que como el fondo del mundo virtual no se ve afectado por el nodo **Fog**, se lo deberá adecuar expresamente para conseguir un efecto visual coherente.

El campo ***color*** define el color con el que se modelará la niebla, mientras que el campo ***visibilityRange*** hace lo propio con la distancia a partir de la cual los objetos quedan totalmente ocultos por la misma. Un valor para ***visibilityRange*** igual a 0 desactiva la acción de este nodo dentro del mundo virtual.

Como ya se comentó existen dos modelos de niebla “LINEAR” y “EXPONENTIAL”, este valor se especificará haciendo uso del campo ***fogType***.

Los eventos ***set_bind*** e ***isBound*** responderán al comportamiento de la pila de nodos **Fog**, de igual forma como se ha visto para el nodo **Background**, permitiendo seleccionar distintos tipos de niebla desde la pila que las contiene.

En el Listado 76 se presenta el código fuente de un ejemplo de uso del nodo **Fog** en el cual se representan ocho cajas separadas medio metro una de la otra y que se ven afectadas por la niebla generada por el nodo **Fog**. Nótese que el color del fondo se ha dispuesto igual que al de la niebla para conservar la uniformidad en la ocultación de las cajas.

```
#VRML V2.0 utf8
Background {
   skyColor [0.4 0.4 0.4]
}
Fog {
   fogType "EXPONENTIAL"
   color 0.4 0.4 0.4
   visibilityRange 3
}
Transform {
   translation 0 0 0
   children Shape {geometry Box {size .1 .1 .1}}
}
```

```
Transform {
   translation 0 0 -0.5
   children Shape {geometry Box {size .1 .1 .1}}
}
Transform {
   translation 0 0 -1
   children Shape {geometry Box {size .1 .1 .1}}
}
Transform {
   translation 0 0 -1.5
   children Shape {geometry Box {size .1 .1 .1}}
}
Transform {
   translation 0 0 -2
   children Shape {geometry Box {size .1 .1 .1}}
}
Transform {
   translation 0 0 -2.5
   children Shape {geometry Box {size .1 .1 .1}}
}
Transform {
   translation 0 0 -3
   children Shape {geometry Box {size .1 .1 .1}}
}
Transform {
   translation 0 0 —3.5
   children Shape {geometry Box {size .1 .1 .1}}
}
```

Listado 76: El nodo Fog

En la Figura 58 se puede contemplar el resultado obtenido a través del listado precedente. Nótese la mezcla del color de los cubos con el de la niebla a medida que se distancian del *avatar*.

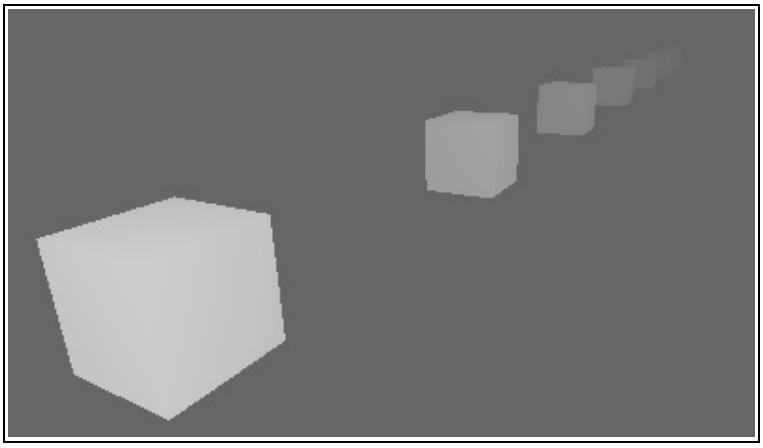


Figura 58: El nodo Fog

3.9.3 Viewpoint

Sintaxis:

```
Viewpoint {
      eventIn      SFBool     set_bind
      exposedField SFFloat    fieldOfView    0.785398  # (0,PI)
      exposedField SFBool     jump           TRUE
      exposedField SFRotation orientation    0 0 1 0   # [-1,1],(-∞,∞)
      exposedField SFVec3f    position       0 0 10    # (-∞,∞)
      field        SFString   description    ""
      eventOut     SFTime     bindTime
      eventOut     SFBool     isBound
}
```

El nodo **Viewpoint** se utiliza para definir las distintas cámaras preestablecidas para el mundo virtual a partir de las cuales se comenzará a desplazar el *avatar*. Este nodo también tiene asociada una pila por lo que contará con los eventos ***set_bind*** e ***isBound***.

El campo ***position*** define la posición inicial para la cámara dentro del sistema de coordenadas local, mientras que el campo ***orientation*** determina la dirección hacia donde estará orientada la visión. Nótese que el vector que define el eje de giro para este campo debe tener sus componentes normalizadas (Véase la Nota 24).

El campo ***description*** indica una cadena de texto que será utilizada dentro de la interfaz de navegación del *plug-in* permitiendo al usuario movilizarse entre las cámaras de forma cómoda y precisa.

Con el campo ***jump*** se controlará si al cambiar de vista el *avatar* modificará su posición y orientación actual (TRUE) o bien no sufrirá cambio alguno (FALSE). Este campo es útil para los casos en que se cambia de vista por medio del evento ***set_bind*** para, por ejemplo, desplazar al *avatar* automáticamente junto con la vista a la cual se lo ha ligado.

El campo ***fieldOfView*** será el encargado de determinar las características de la lente de una vista a través de la cual se observa el mundo virtual. El valor predeterminado (0.785398) presenta cierta similitud con el ojo humano, un valor menor producirá efectos similares al teleobjetivo de una cámara y un valor mayor se asemejará a una lente de gran angular.

El evento de salida ***bindTime*** se producirá cada vez que la vista se coloca en la cima de la pila tornándose activa y también cuando abandone esta condición, dejando la cima. Los eventos ***set_bind*** e ***isBound*** se comportarán de igual forma que con los nodos **Background** y **Fog**.

El Listado 77 contiene el código fuente de un ejemplo de uso del nodo **Viewpoint**. Nótese que las cuatro cámaras definidas contienen distintos valores para el campo ***fieldOfView*** de esta manera se varía la profundidad focal de la escena, provocando deformaciones para valores muy elevados.

```
#VRML V2.0 utf8
Viewpoint {
   fieldOfView 3
   orientation 0 1 0 0
   position 0 0 6
}
Viewpoint {
   fieldOfView 1.5
   orientation 0 1 0 0
   position 0 0 6
}
Viewpoint {
   orientation 0 1 0 0
   position 0 0 6
}
Viewpoint {
   fieldOfView 0.1
```

```
   orientation 0 1 0 0
   position 0 0 6
}
Shape {
   appearance Appearance {material Material {diffuseColor 0 .2 .7}}
   geometry Box { }
}
Transform {
   translation 4 0 0
   children
      Shape {
         appearance Appearance {
            material Material {diffuseColor 0 .7 .2}
         }
         geometry Sphere { }
      }
}
Transform {
   translation -4 0 0
   children
      Shape {
         appearance Appearance {
            material Material {diffuseColor .7 .2 .2}
         }
         geometry Sphere { }
      }
}
```

Listado 77: El nodo Viewpoint

En la Figura 59 se pueden observar los resultados obtenidos para cada vista. Téngase en cuenta que sólo se representa el aspecto de las primitivas, ya que un cambio en el campo ***fieldOfView*** trae aparejado también un acercamiento de la escena si toma valores pequeños, y un alejamiento para valores grandes.

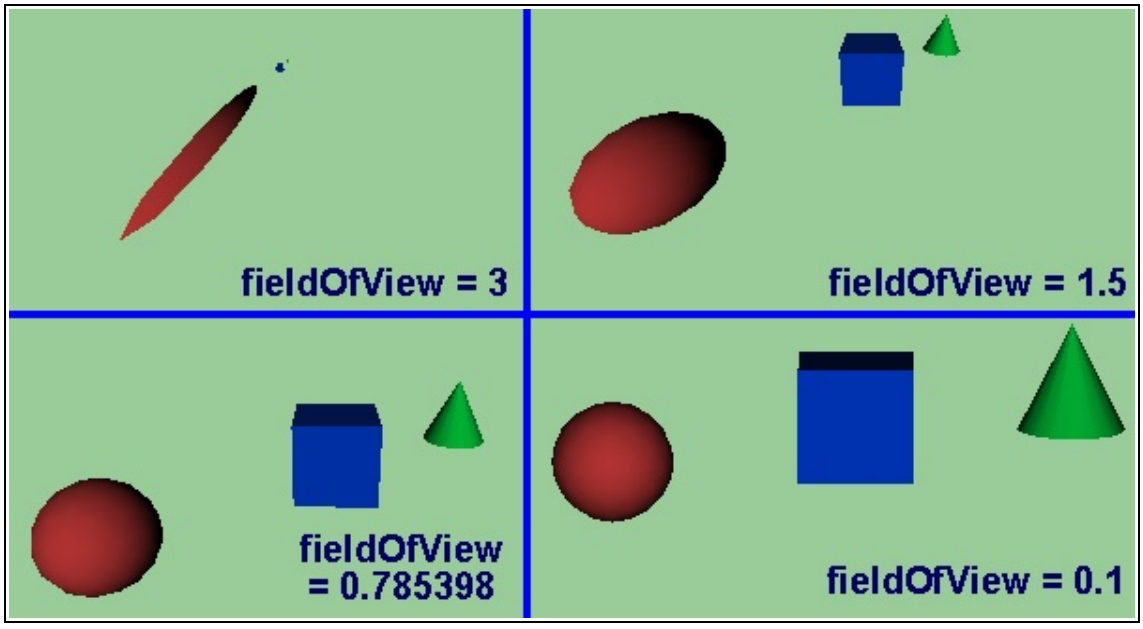


Figura 59: El nodo Viewpoint

Como último apunte para este nodo, tal como se ha mencionado en la descripción del nodo **Anchor** (véase el Punto 3.7.5), se podrá especificar la vista inicial a emplear junto a la *url* al cargar el mundo virtual, o bien dentro del mismo mundo, indicar un salto hacia otra vista.

Para poder utilizar una vista dentro de una *url* será necesaria una referencia a la misma lo que en VRML se consigue mediante la palabra reservada **DEF** seguida del alias particular (véase el Punto 3.10.1).

En el Listado 78 se presenta un ejemplo en el que se ha asignado el alias "Vista" al nodo **Viewpoint**. No se debe confundir la referencia "Vista" con la descripción de la vista que presentará el *plug-in*: "Camara particular". Ahora cuando se quiera referenciar a la cámara del ejemplo desde una *url*, tanto desde un nodo VRML como desde un navegador de Internet, se deberá escribir: "nombredelfichero.wrl#Vista".

```
#VRML V2.0 utf8
DEF Vista Viewpoint {
   orientation 0 1 0 1.5708
   position 0 2 1
   description "Camara particular"
}
```

Listado 78: Referencia a un nodo Viewpoint

3.9.4 NavigationInfo

Sintaxis:

```
NavigationInfo {
      eventIn      SFBool   set_bind
      exposedField MFFloat  avatarSize      [0.25, 1.6, 0.75] # [0,∞)
      exposedField SFBool   headlight       TRUE
      exposedField SFFloat  speed           1.0               # [0,∞)
      exposedField MFString type            ["WALK", "ANY"]
      exposedField SFFloat  visibilityLimit 0.0               # [0,∞)
      eventOut     SFBool   isBound
}
```

El nodo **NavigationInfo** define las características del *avatar*, los tipos de exploración permitidos y la visibilidad (no confundir con niebla) para un mundo virtual. Este nodo también tiene la virtud de contar con una pila de modo que contendrá a los eventos ***set_bind*** e ***isBound***.

El campo ***avatarSize***, de tres valores, define las dimensiones del *avatar* al que se lo puede pensar, desde el punto de vista de las colisiones, como un cilindro de radio igual al primer valor de este campo, altura igual al segundo valor y que es capaz de sortear desniveles más bajos que el tercero de los valores (como si fuesen escalones).

La velocidad de movimiento del *avatar* por el mundo virtual vendrá definida por el campo ***speed***, cuyo valor está expresado en metros por segundo.

El campo ***headlight*** determinará el estado inicial de la luz que lleva el *avatar* a modo de casco de minero. Un valor de TRUE, equivaldrá a una luz encendida y un valor de FALSE, a apagada. Incluso en algunos *plug-ins* FALSE implicará que la luz está inhabilitada.

Con el campo ***type*** se restringirán los modos de navegación que se permitirán dentro del mundo virtual, utilizando una lista ordenada con las palabras que representan a cada uno de los tipos permitidos que podrán ser:

- **ANY:** Permite elegir al *plug-in* el modo que más crea conveniente y que luego el usuario podrá cambiar manualmente.
- **WALK:** El modo más recomendado para explorar un escenario real. El *avatar* experimentará los efectos de la gravedad y las colisiones con las primitivas del mundo, salvo el caso de los nodos **IndexedLineSet**, **PointSet** y **Text.**
- **FLY:** Este modo es similar a WALK, pero con la posibilidad de desactivar a voluntad la gravedad.
- **EXAMINE:** Se utiliza principalmente para examinar un objeto permitiendo acercarlo y alejarlo, rotarlo, e incluso atravesarlo sin que se detecten las colisiones. Tampoco existe la gravedad y en la mayoría de los *plug-ins* se podrá iniciar una rotación arbitraria del mundo, el cual permanecerá en este estado (girando) hasta que se lo detenga expresamente.
- **NONE:** Se elimina todo tipo de navegación a través de los controles provistos por el *plug-in* e incluso su interfaz de mandos. La única manera de desplazarse es mediante el propio código VRML por medio de eventos y enlaces.

Debido a que los *plug-ins* podrán implementar sus propios modos de navegación y a que no todos los modos vistos podrán estar disponibles, mediante la lista ordenada de modos de navegación se escogerá el más apropiado, respetando el orden especificado en el código.

El campo ***visibilityLimit*** indicará la distancia en metros a partir de la cual no se renderizará el mundo virtual. Este campo será utilizado principalmente para aliviar la carga computacional de un *plug-in*.

Los campos ***avatarSize***, ***speed*** y ***visibilityLimit*** sufrirán las trasformaciones escalares que le sean aplicadas a la cámara actualmente activa (nodo **Viewpoint**), mediante los efectos de uno o varios nodos **Transform**.

En el Listado 79 se presenta un ejemplo de uso del nodo **NavigationInfo** que modela al de una persona de 1.70 m de altura, capaz de subir escalones de 35 cm, sujeto a las leyes de la gravedad y que camina velozmente.

```
#VRML V2.0 utf8
NavigationInfo {
   avatarSize [ 0.25, 1.7, 0.35 ]
   headlight FALSE
   type "WALK"
   speed 2
}
```

Listado 79: El nodo NavigationInfo

3.9.5 WorldInfo

Sintaxis:

```
WorldInfo {
      field MFString info  []
      field SFString title ""
}
```

El nodo **WorldInfo** cumple una función informativa dentro del mundo virtual. El campo ***title*** se podrá utilizar para especificar el título del mundo virtual y el campo ***info*** contendrá una lista de cadenas de caracteres con la información extra que se quiera ofrecer. Nótese que los *plug-ins* pueden ignorar la información provista por este nodo.

En el Listado 80 se presenta un ejemplo de uso del nodo **WorldInfo**.

```
#VRML V2.0 utf8
WorldInfo {
   title "EL LENGUAJE VRML97"
   info   ["Daniel Héctor Stolfi Rosso", "Año 2004-2009"]
}
```

Listado 80: El nodo WorldInfo

3.10 Palabras reservadas

Complementando a los nodos vistos, existen en VRML una serie de palabras reservadas destinadas a facilitar el uso de los nodos mediante referencias y a ampliar la funcionalidad propia del lenguaje.

3.10.1 DEF y USE

En varios de los ejemplos vistos en este capítulo, cuando se necesitaba una referencia a un nodo, se ha hecho uso de la palabra reservada **DEF**. Pero la funcionalidad que ofrece esta palabra es más amplia aún, especialmente cuando se la combina con su palabra complementaria **USE**.

Por ejemplo, tomando el Listado 81 se define una referencia al nodo **Group** llamada “Linea” mediante la palabra reservada **DEF**, en su interior como primer hijo se modela una caja de color verde creando también una referencia a la misma llamada “Caja”. Ahora dentro del grupo se utilizará la referencia “Caja” dos veces por medio de la palabra reservada **USE**, afectada por sendas traslaciones. De esta manera se obtiene un grupo llamado “Linea” que contiene a tres cajas de color verde.

Posteriormente mediante el uso de dos nodos **Transform** se modelan dos líneas de tres cajas más, desplazadas hacia arriba 4 y 8 metros respectivamente pero en vez de definir cada línea nuevamente, se utiliza la referencia al grupo previamente creada y que se la ha llamado “Linea”, haciendo uso de la palabra reservada **USE**.

```
#VRML V2.0 utf8
DEF Linea Group {
   children [
      DEF Caja Shape {
         appearance Appearance {
            material Material {diffuseColor 0 0.8 0}}
         geometry Box { }
      }
      Transform {
         translation 4 0 0
         children USE Caja
      }
      Transform {
         translation -4 0 0
         children USE Caja
      }
   ]
}
Transform {
   translation 0 4 0
   children USE Linea
}
Transform {
   translation 0 8 0
   children USE Linea
}
```

Listado 81: Las palabras reservadas DEF y USE

El resultado obtenido se puede observar en la Figura 60. Nótese que si se cambia alguna propiedad de la primitiva, como por ejemplo el color, se verán afectadas por el cambio las nueve cajas que componen al mundo virtual.

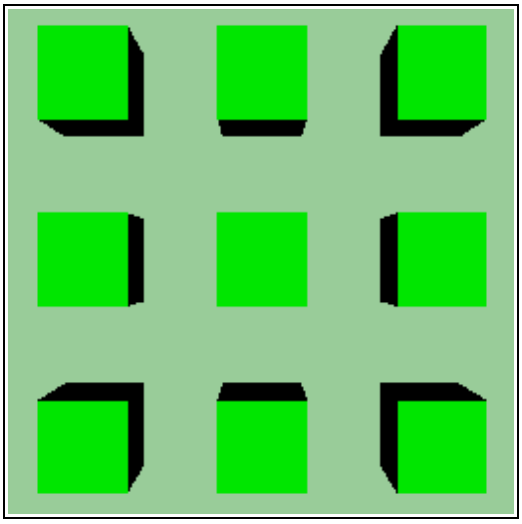


Figura 60: Las palabras reservadas DEF y USE

Como se desprende del párrafo anterior el uso de **DEF** y **USE** puede resultar un arma de doble filo. Las entidades definidas con **DEF** no se copian cuando se las vuelve a utilizar mediante **USE**, sino que se renderiza la misma entidad sujeta a las transformaciones que se hayan especificado. Si se trata de una entidad activa, cuando varíe el estado del original en realidad variará en todas las apariciones de la misma.

Además de para poder ser utilizada junto con **USE,** otra utilidad de la palabra **DEF** es la de dotar a un nodo de un "alias" para luego ser referenciado por la palabra reservada **ROUTE** como se verá a continuación. Si se trata de nodos **Viewpoint**, el nombre dado podrá utilizarse junto con la *url* desde donde se carga el mundo para indicar la vista inicial (véase el nodo **Anchor** en el Punto 3.7.5 y el nodo **Viewpoint** en el Punto 3.9.3).

3.10.2 ROUTE y TO

La palabra reservada **ROUTE** se encargará de dirigir un evento de salida proveniente de un nodo hacia (**TO**) un evento de entrada perteneciente a otro nodo. Véase en los ejemplos para los nodos de los Puntos 3.7 y 3.8 (sensores e interpoladores) el modo de empleo de estas palabras reservadas.

3.10.3 PROTO, IS y EXTERNPROTO

La palabra reservada **PROTO** se utiliza para definir un prototipo, el cual no es más que un nuevo nodo creado en base a las primitivas propias de VRML. Dentro de la definición se podrán especificar los campos normales, los campos públicos y los eventos que pertenecerán al nuevo nodo creado junto con su tipo y su valor por defecto.

En el Listado 82 se presenta un ejemplo de definición de un prototipo que se representa a una pared de ladrillos.

```
#VRML V2.0 utf8
PROTO Pared [
   field SFVec3f  tam   2 2 2
   field SFVec2f  esc   1 1
   field SFVec2f  tra   0 0
]
{
   Shape {
      appearance Appearance {
         material Material {ambientIntensity 0.6}
         texture ImageTexture {url "pared_ladrillos.gif"}
         textureTransform TextureTransform {scale IS esc
   translation IS tra}
      }
      geometry Box {size IS tam}
   }
}
```

Listado 82: Las palabras reservadas PROTO e IS

Observando el ejemplo, luego de la palabra reservada **PROTO** aparece el nombre que se le dará al prototipo seguido de la lista de campos y/o eventos encerrados entre corchetes. Cada campo (o evento) irá seguido de su tipo (véase en la Tabla 1 los tipos presentes en VRML), del nombre para el campo y de su valor por defecto.

A continuación encerrado entre llaves, irá el código VRML que hará uso de los campos y/o eventos de la definición, precediendo a su nombre con la palabra reservada **IS**.

De este modo se logrará modelar de forma sencilla a una pared de ladrillos de, por ejemplo 10 x 2 x 2 metros, con sólo escribir: **Pared (tam** 10 2 2**)**. Téngase en cuenta que los campos que se omitan tomarán el valor predeterminado especificado en la definición del prototipo.

Si el prototipo se encuentra definido en un fichero distinto al que lo intenta utilizar (estrategia recomendada para mantener separados en un fichero a todos los prototipos utilizados), antes de que se lo pueda emplear se deberá utilizar la palabra reservada **EXTERNPROTO** para crear una referencia al fichero en donde se encuentra la definición del prototipo.

Como ejemplo de utilización de la palabra reservada **EXTERNPROTO** se presenta el Listado 83 en donde se define al prototipo "Pared" con los campos listados entre corchetes, como el prototipo del mismo nombre ubicado en el fichero *prototipos.wrl*.

Nótese que por motivos de coherencia en los nombres se ha utilizado el mismo para el prototipo en el fichero actual que el del fichero que contiene a todos los prototipos; pero la sintaxis permite el renombrado e incluso la elección de un subconjunto de campos que estarán disponibles sólo para el fichero actual.

```
#VRML V2.0 utf8
EXTERNPROTO Pared [field SFVec3f tam field SFVec2f esc
           field SFVec2f tra] "prototipos.wrl#Pared"
Transform {
   translation 4 0 0
   children Pared {tam 1 2 1 esc 2 0.5 tra 0.5 0.5}
}
```

Listado 83: La palabra reservada EXTERNPROTO

Es importante aclarar que no sólo se pueden hacer prototipos para las primitivas, también es posible definir prototipos para materiales. De esta manera se consigue centralizar las definiciones de los mismos en un fichero (*materiales.wrl* por ejemplo) para luego ser utilizados dentro del resto de ficheros que componen el modelo.

El Listado 84 presenta un ejemplo de uso para los prototipos de los materiales. El listado superior obedece a una porción del fichero *materiales.wrl* en donde se encuentran las definiciones. Mientras que el inferior presenta un ejemplo de uso de los mismos dentro de un fichero independiente.

```
#VRML V2.0 utf8
PROTO MAT_viga    [] {Material {diffuseColor 0.816 0.8 0.648
                               ambientIntensity 0.5}}
PROTO MAT_opaco   [] {Material {ambientIntensity 0.6}}
PROTO MAT_blanco  [] {Material {ambientIntensity 0.5
                               diffuseColor 1 1 1}}
PROTO MAT_naranja [] {Material {ambientIntensity 0.5
                               diffuseColor 0.891 0.773 0.484}}
```

```
#VRML V2.0 utf8
EXTERNPROTO MAT_blanco  [] "materiales.wrl#MAT_blanco"
EXTERNPROTO MAT_opaco   [] "materiales.wrl#MAT_opaco"

Shape {
```

```
   appearance Appearance {material MAT_blanco {}}
   geometry IndexedFaceSet {
      coord Coordinate {point [-1 0 0, 0 1 0, 1 0 0]}
      coordIndex [0,1,2,-1]
   }
}
Shape {
   appearance Appearance {material MAT_naranja {}}
   geometry Box {size 1 0.5 1}
}
```

Listado 84: Uso de prototipos para los materiales

3.11 Script

Sintaxis:

```
Script {
      exposedField MFString url           []
      field        SFBool   directOutput  FALSE
      field        SFBool   mustEvaluate  FALSE

      eventIn      tipoEvento nombreEvento                # uno por cada evento de entrada
      field        tipoCampo  nombreCampo  valorInicial   # uno por cada campo
      eventOut     tipoEvento nombreEvento                # uno por cada evento de salida
}
```

Dada la complejidad que reviste el nodo **Script** su descripción ha sido pospuesta hasta este punto habiendo visto primero las primitivas, las técnicas de enrutado, el uso de eventos y los prototipos. Por lo tanto el lector debería comprender las siguientes líneas sin mayores dificultades.

La principal función del nodo **Script** es la de dotar a los mundos virtuales de vida. Si bien como se ha visto con los sensores e interpoladores se conseguía cierto dinamismo, cuando se necesitan tomar decisiones, realizar repeticiones o guardar información, la única opción es aprovechar las bondades que ofrece el nodo **Script**.

El código del *script* en sí, será referenciado desde el campo ***url*** ofreciéndose la posibilidad incluso de escribir el propio código dentro de este campo, sin la necesidad de utilizar una referencia a un fichero externo. El lenguaje admitido es ECMAScript[30], pero para los ejemplos de este proyecto se utilizará un subconjunto del mismo llamado *vrmlscript*.

Si bien el objetivo de este punto es explicar la sintaxis y forma de uso del nodo **Script**, se darán nociones de *vrmlscript* para permitir la comprensión de los ejemplos. Debido a que el profundizar sobre el lenguaje ECMAScript requeriría un libro dedicado, queda a voluntad del lector ampliar los conocimientos sobre este lenguaje a través de la bibliografía disponible.

Los eventos de entrada que se definan dentro del nodo **Script** se corresponderán dentro del código con funciones; los eventos de salida se representarán con variables, y los campos definidos e inicializados por

[30] Lenguaje que tiene sus orígenes en JavaScript, el cual fue propuesto para ser estandarizado por Netscape Communications en Otoño de 1996. Actualmente la edición cuarta de ECMAScript es un subconjunto de JavaScript 2.0.

el nodo **Script**, serán las variables globales del código del *script*. Además del valor que provee el evento, una función siempre dispondrá como segundo parámetro de la hora de producción del mismo.

Como ejemplo de uso se tiene el Listado 85 que simula el comportamiento de un tapiz de proyección (Figura 61). Nótese que no se han incluido primitivas para tener un código más limpio. Se puede consultar el código fuente completo en el Punto 5.1.

```
#VRML V2.0 utf8
DEF TAPIZ Transform {
   translation 0 -.95 -.01
   scale 1 .15 1
   center 0 .95 0
   children    # PRIMITIVAS PARA EL TAPIZ
}
DEF CORDEL Transform {
   translation 0 -.5 -.01
   children [
      # PRIMITIVA PARA EL CORDEL
      DEF TS TouchSensor { }
   ]
}
DEF SC Script {
   eventIn  SFTime   toque
   eventOut SFVec3f  escala
   eventOut SFVec3f  desplaz
   field  SFBool     bajado   FALSE
   url "vrmlscript:
      function toque() {
         if (bajado) {
            bajado = FALSE;
            escala = new SFVec3f(1,.15,1);
            desplaz = new SFVec3f(0,-.5,-.01);
         }
         else {
            bajado = TRUE;
            escala = new SFVec3f(1,1,1);
            desplaz = new SFVec3f(0,-2.1,-.01);
         }
      }
   "
}
ROUTE TS.touchTime TO SC.toque
ROUTE SC.escala TO TAPIZ.set_scale
ROUTE SC.desplaz TO CORDEL.set_translation
```

Listado 85: Script para el tapiz de proyección

El *script* apodado "SC" define un evento de entrada llamado ***toque*** que es de tipo SFTime, dos eventos de salida ***escala*** y ***desplaza*** de tipo SFVec3f y un campo ***bajado*** inicializado a FALSE que será el utilizado para conocer en todo momento si el tapiz está enrollado (*bajado* = FALSE) o desenrollado (bajado = TRUE).

La función ***toque*** se activará cuando llegue un evento proveniente del sensor de toque "TS", si el tapiz está enrollado (*bajado* = FALSE), se cambiará a *bajado* = TRUE y se emitirá un evento de salida ***escala*** con el valor creado con el constructor SFVec3f. Este evento será enrutado al nodo **Transform** cuyo alias es "TAPIZ" por medio de la palabra reservada ROUTE (al final del fichero) para cambiar la escala de representación del tapiz y presentarlo a tamaño completo. Además se genera otro evento de salida

desplaz con las nuevas coordenadas para el cordel, el cual debe desplazarse para mantener su posición respecto al tapiz. Este evento será enrutado hacia el nodo **Transform** apodado "CORDEL" el cual realizará la traslación requerida.

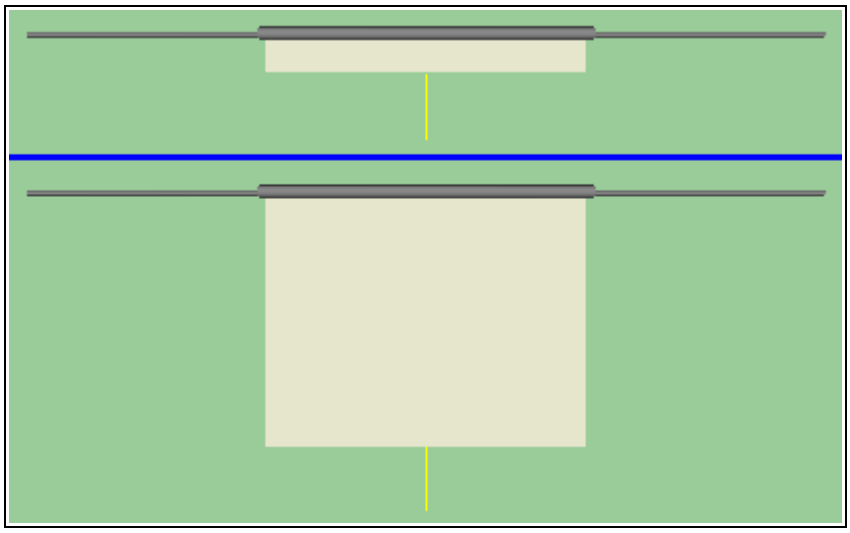


Figura 61: Tapiz recogido y extendido

Por el contrario si el tapiz estaba desenrollado (*bajado* = TRUE), las acciones llevadas a cabo serán cambiar el valor de *bajado* a FALSE, aplicar nuevamente la escala al tapiz reduciendo su longitud para que se presente enrollado y desplazar el cordel para situarlo acorde a la nueva situación del tapiz.

Vrmlscript provee además de una biblioteca de constantes y funciones matemáticas, heredadas de JavaScript, y por lo tanto se las encuentra como métodos de la clase **Math**. Para su uso se deberán invocar junto con el nombre de la clase, por ejemplo: **e = Math.E** asigna a la variable "e" el valor de la constante de Euler. En la Tabla 4 se presenta el listado de constantes y métodos del objeto **Math**.

Constante	**Descripción**
E	Constante de Euler
LN10	Logaritmo neperiano de 10
LN2	Logaritmo neperiano de 2
PI	Constante PI
SQRT1_2	Raíz cuadrada de 1/2
SQRT2	Raíz cuadrada de 2
Métodos	**Descripción**
abs(número)	Devuelve el valor absoluto de *número*
acos(número)	Devuelve el ángulo en radianes cuyo coseno es *número*
asin(número)	Devuelve el ángulo en radianes cuyo seno es *número*
atan(número)	Devuelve el ángulo en radianes cuya tangente es *número*
ceil(número)	Devuelve *número* redondeado al entero superior más próximo

cos(número)	Devuelve el coseno de *número* que debe estar en radianes
exp(número)	Devuelve e (constante de Euler) elevado a *número*
floor(número)	Devuelve *número* redondeado al entero inferior más próximo
log(número)	Devuelve el logaritmo neperiano de *número*
max(número1, número2)	Devuelve el mayor entre *número1* y *número2*
min(número1, número2)	Devuelve el menor entre *número1* y *número2*
pow(base, exponente)	Devuelve *base* elevado a *exponente*
random()	Devuelve un número aleatorio entre 0 y 1
round(número)	Devuelve número redondeado al entero más próximo
sin(número)	Devuelve el seno de *número* que debe estar en radianes
sqrt(número)	Devuelve la raíz cuadrada de número
tan(número)	Devuelve la tangente de *número* que debe estar en radianes

Tabla 4: Constantes y métodos del objeto Math

Además existe un objeto Browser que proveerá de información sobre el entorno donde se está ejecutando el mundo virtual. En la Tabla 5 se presenta el listado de constantes y métodos del objeto **Browser**.

Métodos
String getName() Devuelve una cadena de caracteres con el nombre del *plug-in.*
String getVersion() Devuelve una cadena de caracteres con la versión del *plug-in.*
SFFloat getCurrentSpeed() Devuelve el número en coma flotante con la velocidad actual a la que el *avatar* se está desplazando por la escena.
SFFloat getCurrentFrameRate() Devuelve el número en coma flotante con el número de *FPS* actuales con los que se está renderizando la escena.
String getWorldURL() Devuelve una cadena de caracteres con la *url* desde la cual se ha cargado al mundo virtual.
void replaceWorld(MFNode lista_de_nodos) Reemplaza el mundo actual por la lista de nodos pasada como parámetro.
MFNode createVrmlFromString(String código_vrml) Convierte la cadena de caracteres pasada como parámetro en un vector de nodos.
void createVrmlFromURL(MFString url, Node nodo, String evento) Convierte la *url* pasada en un mundo virtual y a continuación envía el evento al nodo especificado.
void addRoute(SFNode nodo_O, String evento_O, SFNode nodo_D, String evento_D) Crea una ruta (**ROUTE**) para el evento origen (evento_O) del nodo origen (nodo_O) hacia (**TO**) el evento destino (evento_D) del nodo destino (nodo_D).
void deleteRoute(SFNode nodo_O,String evento_O, SFNode nodo_D, String evento_D)

Destruye la ruta para el evento origen (evento_O) del nodo origen (nodo_O) hacia el evento destino (evento_D) del nodo destino (nodo_D), siempre que la misma exista.
void loadURL(MFString url, MFString parámetro) Carga el mundo desde la *url* especificada, en parámetro se puede especificar un marco de destino. Si se carga en el marco actual este método nunca retorna.
void setDescription(String descripción) Coloca como descripción (normalmente en la barra de estado del navegador web) la cadena pasada como parámetro. El comportamiento es similar al campo ***description*** del nodo **Anchor**.

Tabla 5: Métodos del objeto Browser

Por último, existen dos funciones disponibles una que se ejecuta al cargarse el *script* llamada **initialize()** y otra llamada **shutdown()** que se ejecuta al salir del mundo que contiene al *script*. En el Listado 86 se presenta un ejemplo de uso de la función **initialize()**, en donde se la utiliza para imprimir en la consola que normalmente proveen los *plug-ins* el nombre y la versión del *plug-in* que se está empleando.

```
#VRML V2.0 utf8
Script {
   url "vrmlscript:
      function initialize() {
         print('plug-in:' + Browser.getName());
         print('version:' +   Browser.getVersion());
      }
   "
}
```

Listado 86: Ejemplo de uso de la función initialize()

Queda por explicar el campo ***directOutput*** del nodo **Script** que controla la posibilidad de que el *script* pueda enviar eventos directamente a otros nodos, y que pueda establecer y remover rutas a voluntad. El valor por defecto (FALSE) impide este comportamiento limitando la acción del *script* sólo a través de sus eventos de salida que serán enrutados utilizando la palabra reservada **ROUTE**.

Y por último, el campo ***mustEvaluate***, que define la forma que el *plug-in* envía eventos al nodo **Script**. Si este campo toma el valor FALSE (por defecto), el envío de eventos hacia el *script* se postergará hasta que se necesiten los eventos de salida que éste produce. Mientras que si el valor de este campo es TRUE, los eventos le serán enviados lo antes posible, independientemente de que se necesiten sus salidas. Esta opción, como se puede intuir, degrada notablemente el rendimiento global del mundo virtual pero puede ser útil cuando las salidas del *script* estén dirigidas a un entorno externo al *plug-in*.

3.12 Nurbs (Non-Uniform Rational B_Spline)

Según el estándar de VRML97 los navegadores y *plug-ins* VRML no están obligados a dar soporte a este grupo de nodos para cumplir con la especificación, por lo que en este libro sólo se hará una breve descripción sobre los mismos.

Los nodos que pertenecen a esta categoría se utilizan para definir entidades en base a curvas y superficies permitiendo obtener una precisión mucho mayor que la lograda con las primitivas tradicionales de VRML.

Debido a la complejidad que reviste la construcción de un modelo mediante estos nodos, normalmente los modelos se crearán utilizando algún editor específico o importándolos desde algún programa de CAD/CAM.

Los nodos que define el estándar son los siguientes:

- Contour2D
- CoordinateDeformer
- NurbsCurve
- NurbsCurve2D
- NurbsGroup
- NurbsPositionInterpolator
- NurbsSurface
- NurbsTextureSurface
- Polyline2D
- TrimmedSurface

3.13 Nodos Geoespaciales

Esta categoría de nodos, tampoco está soportada actualmente por todos los *plug-ins* y al igual que con los nodos **Nurbs**, no es requisito indispensable el dar una implementación de los mismos para cumplir con el estándar de VRML97.

Los nodos Geoespaciales consisten en un grupo de nodos en los que sus coordenadas vienen expresadas como referencias a coordenadas terrestres (longitud, latitud y elevación). Por este motivo son empleados principalmente en modelos cartográficos y en mundos donde se modelan grandes extensiones de terreno permitiendo contemplar con exactitud la orografía del mismo.

Los nodos que define el estándar son los siguientes:

- GeoCoordinate
- GeoElevationGrid

- GeoLocation
- GeoLOD
- GeoMetadata
- GeoOrigin
- GeoPositionInterpolator
- GeoTouchSensor
- GeoViewpoint

GeoVRML 1.1 es una implementación de nodos Geoespaciales realizada en Java y de código abierto, puede descargarse de **http://www.geovrml.org/** y se instala como una extensión para el *plug-in* de VRML que ya debe existir en el sistema.

3.14 Resumen

A lo largo de este capítulo se han abordado la totalidad de los nodos que conforman el estándar de VRML97. Mediante ejemplos y figuras se intentó dar una explicación específica de cada nodo, particularizando el comportamiento de cada campo.

Se comenzó el estudio con los nodos que modelan primitivas para obtener representaciones en tres dimensiones del código que contiene cada uno de ellos, a continuación se fue dotando de colores y texturas al modelo especificando el material con el que se construían.

Posteriormente se introdujo el concepto de iluminación y los distintos tipos de nodos que representan a las luces del mundo virtual, acto seguido mediante el sonido se entró en el aspecto multimedia de la realidad virtual.

Con los nodos de agrupación se ha aprendido a realizar transformaciones sobre un conjunto de nodos y algunas técnicas de optimización, para luego con los sensores e interpoladores conseguir que el mundo realmente cobre vida permitiendo que el visitante interactúe con su entorno y reciba respuestas a sus acciones.

Finalmente con los nodos de control del entorno, se pudo modificar las características propias de la navegación, la interfaz que brinda el *plug-in* al usuario y el aspecto del fondo del mundo virtual.

Un apartado propio ha merecido el nodo **Script** para describir su forma de uso y dar unas nociones sobre el lenguaje de programación *vrmlscript*.

A continuación, ahora que VRML ya no es un lenguaje desconocido para el lector, abordaremos las mejoras que se pueden realizar sobre el código de un mundo complejo, optimizando las representaciones de las entidades y procesando los ficheros mediante programas construidos por medio de un procesador lexicográfico, obteniéndose un código más compacto, más eficiente y con las mismas características que el original.

Capítulo 4

Optimizaciones

4.1 Descripción del Problema

Una vez modelado un mundo virtual de gran tamaño, uno descubre ciertos problemas de fluidez en el movimiento debidos principalmente al renderizado constante de objetos ricos en detalles, incluso cuando no se encuentran a la vista o cuando están demasiado distantes como para que se pueda apreciar sus particularidades debido principalmente a la resolución limitada de la pantalla. Otro problema que se puede advertir es que el tamaño total de almacenamiento ha crecido vertiginosamente, en especial por la inclusión de ficheros para los sonidos y texturas.

Debido a que la mayoría de los desarrollos tienen como objetivo ser visualizados en un amplio abanico de computadores y en muchos casos a través de Internet, se debe mantener bajo el nivel de carga computacional y a la vez minimizar el tamaño final de los ficheros que serán transmitidos por la red.

Por estos motivos surgen las siguientes dudas: ¿para qué renderizar el interior de habitaciones cuando se está fuera de ellas?. O ¿para qué dibujar con lujo de detalles un edificio cuando nos encontramos en el otro extremo del mundo?. E incluso ¿para qué renderizar unas rosas cuando estamos tan lejos que sólo se ve un punto rosa en la pantalla?.

En respuesta a estos problemas e interrogantes se abordarán las siguientes técnicas de optimización desglosadas en puntos diferenciados según el carácter de las mismas.

4.2 Reducción de detalles con la distancia

La técnica de reducción progresiva de detalles a implementar, que llegará en algunos casos hasta el borrado total del objeto, hace uso del nodo **LOD**. Este nodo, que se ha analizado detalladamente en el Punto 3.6.8, cumplirá la función de seleccionar la representación más apropiada a medida que nos acerquemos o alejemos de los objetos afectados por el mismo.

Por ejemplo, un conjunto de mesas y sillas con computadores constará de tres niveles de detalle: Entre los 0 y 20 metros se presentará en toda su plenitud, entre los 20 y 30 sólo se modelará una caja reemplazando a las mesas y a los computadores y a partir de los 30 metros ya no se renderizará nada ahorrándose un considerable número de polígonos.

En la Figura 62 se observan los tres pasos en la reducción de detalles para la mesa de computadores.

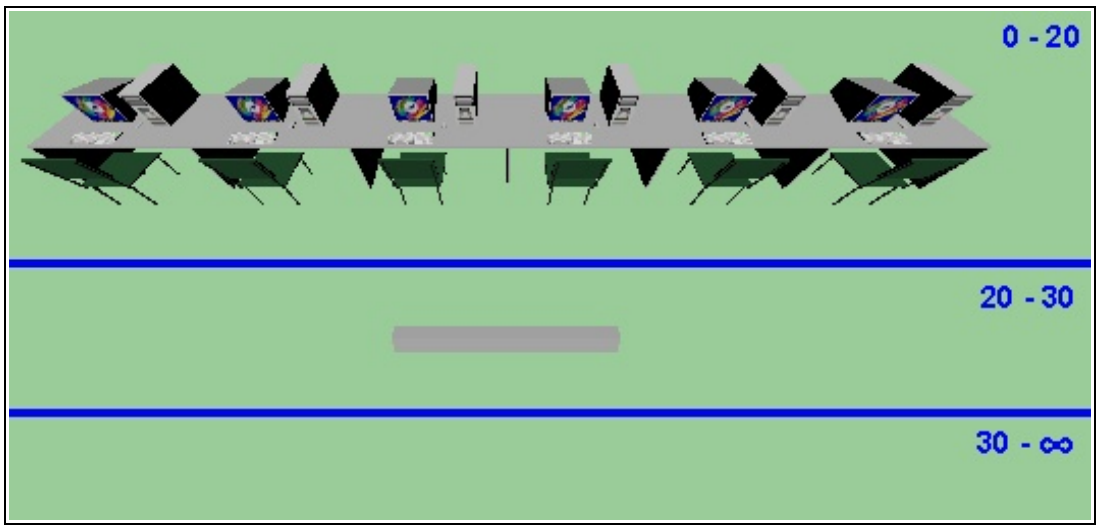


Figura 62: Optimización de la mesa de computadores

Esta misma técnica se puede aplicar en un sinnúmero de ocasiones, aumentando el trabajo del desarrollador pero optimizando notablemente el mundo virtual. En general todo mobiliario debería ser modelado de forma tal que a cierta distancia se prescinda del mismo al estar fuera del campo visual del *avatar* o por la pérdida de calidad en su representación debida a la resolución finita de la pantalla.

Mediante el empleo de esta técnica se ha conseguido disminuir notablemente la cantidad de polígonos simultáneos en escena, pagando el precio de perder detalles a la distancia o de que el mobiliario y otros complementos surjan de repente a medida que el *avatar* se acerca a su emplazamiento.

4.3 Reducción del tamaño de los ficheros de imágenes y sonidos

De cara a minimizar el tamaño de los ficheros de imágenes y sonidos a transferir por Internet, se adoptará una situación de compromiso entre vistosidad y tamaño, de manera que la degradación de la calidad final pase lo más desapercibida posible.

Para las texturas en formato *.GIF* se reducirá la cantidad de colores de la paleta a los estrictamente necesarios, mientras para las de formato *.JPG* se disminuirá la calidad del fichero hasta que la pixelación esté a punto de hacerse evidente.

A los ficheros de sonido se les disminuirá la frecuencia de muestreo, bajando con la misma la calidad pero también el tamaño (es una pena que el estándar no contemple la reproducción de ficheros .MP3 u .OGG). En cuanto a la duración de cada uno, se la reducirá tanto como sea posible aprovechando la opción de reproducción cíclica (***loop***) que ofrece el nodo **AudioClip**, tal como se ha visto en el Punto 3.5.1.1.

4.4 Procesado automático mediante el analizador lexicográfico

Otro aspecto a tener en cuenta es la cantidad de inclusiones que se realizan dentro de un fichero a través del nodo **Inline**. Esta técnica, sumamente útil, que permite incluir mundos virtuales dentro de otros facilitando la organización de los ficheros (véase el Punto 3.6.5), presenta el inconveniente de que cada inclusión obliga al *plug-in* a la apertura de un fichero extra, al cálculo de la caja contenedora (si no se la ha especificado por código) y a decidir cuando es el mejor momento de proceder a la carga.

Eliminando archivos que no son utilizados por más de un fichero y reemplazando la referencia directamente por el código externo se consigue una notable mejora en el consumo de recursos que es apreciable durante la navegación, en dónde se evita la pérdida de fluidez en los movimientos en el momento que es necesario que el *plug-in* lea estos ficheros desde el disco o la red.

Llevando esta técnica al extremo se puede optar por prescindir de los prototipos y reemplazar el uso reiterado de texturas por una referencia a la primera ocurrencia de la misma dentro de cada uno de los ficheros que modelan al mundo virtual.

En las versiones de desarrollo, el uso de prototipos es de inestimable ayuda permitiendo despreocuparse por las características de cada material y disminuyendo enormemente la cantidad de código a escribir. Asimismo el uso de tabuladores y comentarios mantiene el código fuente formateado y fácil de leer. Pero al ser VRML un lenguaje interpretado, no existe compilador por lo que no siempre (casi nunca) el código más claro para el entendimiento humano es el más óptimo para ser procesado por un intérprete.

Por este motivo se construirá haciendo uso del analizador lexicográfico PCLEX[31], un preprocesador desglosado en tres pasadas sucesivas sobre cada fichero fuente *.WRL*, transformándolo progresivamente en un código más eficiente.

Para preparar la optimización será necesario escribir un programa en lenguaje C, que lo llamaremos ***control*** y cuyo código fuente se puede consultar en el Apéndice A. Este programa será el encargado de escribir un *script* de proceso por lotes (fichero *.BAT*) el cual será ejecutado posteriormente por el intérprete de comandos realizando las pasadas de optimización sobre cada uno de los ficheros .WRL.

4.4.1 El programa *control*

Para su correcto funcionamiento *control* precisa leer desde su entrada estándar la lista de ficheros a optimizar separados por un retorno de carro. Haciendo uso del comando del sistema **dir** se obtendrá el listado necesario y mediante una tubería (*pipe*) se enviará su salida hacia la entrada estándar del programa *control*. La línea de comando completa que se ha utilizado es la siguiente:

[31] PCLEX es una marca registrada de Abraxas Software Inc.

```
dir *.wrl /B /ON | control.exe > script.bat
```

El comando **`dir *.wrl /B /ON`** lista a todos los ficheros con extensión *.WRL* del directorio actual, en formato breve (**/B**) y ordenados por nombre (**/ON**). A continuación mediante la tubería (pipe) se envía su salida al programa *control* el cual va escribiendo en su salida estándar los comandos necesarios para realizar las tres pasadas. Mediante la redirección "**>**" se envía la salida de *control* al fichero *script.bat* de forma que luego pueda ser ejecutado por el sistema.

4.4.2 El programa *materiales*

Una vez escrito el fichero *script.bat* y antes de comenzar su ejecución ha sido necesario un paso intermedio. Como en la optimización que se llevará a cabo se trabajará con los prototipos de materiales, es preciso contar con un fichero convenientemente construido, de donde se pueda extraer de forma cómoda la información pertinente a cada material. Para ello se ha escrito en código LEX, el Listado 87 que dará lugar, luego de ser metacompilado, a un código en lenguaje C el que al ser finalmente compilado devendrá en el programa ejecutable *materiales*.

```
%{
/* EL LENGUAJE VRML97
 * DANIEL HECTOR STOLFI ROSSO - 2004-2009
 *
 * programa que extrae los materiales desde la entrada
 * y produce una salida con los mismos formateados convenientemente
 *
 */

%}
%START PROTO DESC
%%
"PROTO "              {BEGIN PROTO;}
<PROTO>"#".+\n           {;}
<PROTO>[A-Za-z_][A-Za-z0-9_]* {printf("%s\n",yytext);BEGIN 0;}
\[\]" {"              {BEGIN DESC;}
<DESC>"#".+\n            {;}
<DESC>.+              {yyless(yyleng-1);
                  printf("%s\n",yytext);BEGIN 0;}
"#".+\n               {;}
.|\n                  {;}
%%
void main(){
   yylex();
}
```

Listado 87: Código fuente LEX para el procesado del fichero materiales.wrl

Nótese que básicamente la labor que el programa desempeña consiste en leer de la entrada estándar el contenido de un fichero con prototipos de materiales (*materiales.wrl* en nuestro ejemplo) y sacar por la salida estándar el nombre de cada prototipo encontrado seguido en la próxima línea por el código VRML que lo representa. Este código se utilizará para reemplazar posteriormente a dicho prototipo en las pasadas de optimización.

El programa será llamado desde la línea de comandos con la siguiente orden:

```
materiales.exe < materiales.wrl > mat.dat
```

Nótese que se dirige a la entrada estándar del programa *materiales* el fichero *materiales.wrl* y la salida del programa se almacenará en el fichero *mat.dat*.

En el Listado 88 se presenta como ejemplo una porción del fichero *materiales.wrl* y en el Listado 89 la porción correspondiente obtenida en el fichero *mat.dat*.

```
#VRML V2.0 utf8
# PROTOTOTIPOS DE MATERIALES
PROTO MAT_viga [] {Material {diffuseColor .816 .8 .648 ambientIntensity .5}}
PROTO MAT_opaco [] {Material {ambientIntensity .6}}
PROTO MAT_blanco [] {Material {ambientIntensity .5 diffuseColor 1 1 1}}
PROTO MAT_naranja [] {Material {ambientIntensity .5 diffuseColor .891 .773 .484}}
PROTO MAT_ladrillo [] {Material {ambientIntensity .5 diffuseColor .856 .535 .340}}
PROTO MAT_aluminio [] {Material {diffuseColor .55 .55 .55 ambientIntensity .5}}
PROTO MAT_cristal [] {Material {diffuseColor .2 .2 .2 emissiveColor 0 0 0 transparency .8}}
....
```

Listado 88: Porción del fichero materiales.wrl

```
MAT_viga
Material {diffuseColor 0.816 0.8 0.648 ambientIntensity 0.5}
MAT_opaco
Material {ambientIntensity 0.6}
MAT_blanco
Material {ambientIntensity 0.5 diffuseColor 1 1 1}
MAT_naranja
Material {ambientIntensity 0.5 diffuseColor 0.891 0.773 0.484}
MAT_ladrillo
Material {ambientIntensity 0.5 diffuseColor 0.856 0.535 0.340}
MAT_aluminio
Material {diffuseColor 0.55 0.55 0.55 ambientIntensity 0.5}
MAT_cristal
Material {diffuseColor .2 .2 .2 emissiveColor 0 0 0 transparency .8}
....
```

Listado 89: Porción del fichero mat.dat

Nótese que el contenido del fichero *mat.dat* no son más que pares, clave-código que luego se utilizarán para realizar los reemplazos dentro de los ficheros *.WRL* a optimizar.

4.4.3 Primera pasada

En la primera pasada, llevada a cabo por el programa *primera*, se eliminarán todos los comentarios del fichero *.WRL*, incluida por cuestiones de simplicidad la cabecera VRML, para posteriormente reemplazar los prototipos de paredes utilizados por el código correspondiente que modela a la primitiva que estos prototipos representaban.

Como ejemplo tómese el Listado 90 en dónde se pueden observar las definiciones de prototipos para paredes haciendo uso de los materiales en *materiales.wrl*.

```
#VRML V2.0 utf8
# PROTOTIPOS PARA LAS PAREDES
EXTERNPROTO MAT_opaco [] "materiales.wrl#MAT_opaco"
EXTERNPROTO MAT_blanco [] "materiales.wrl#MAT_blanco"
EXTERNPROTO MAT_aluminio [] "materiales.wrl#MAT_aluminio"
EXTERNPROTO MAT_cristal [] "materiales.wrl#MAT_cristal"

# PROTOTIPO PARED DE LADRILLOS
PROTO Pared [
   field SFVec3f  tam   2 2 2
   field SFVec2f  esc   1 1
   field SFVec2f  tra   0 0
   ]
   {
      Shape {
         appearance Appearance {
            material MAT_opaco {}
            texture ImageTexture {url "pared_ladrillos.gif"}
            textureTransform TextureTransform {scale IS esc translation IS tra}
         }
         geometry Box {size IS tam}
      }
   }
# PROTOTIPO PARED BLANCA
PROTO Blanca [
   field SFVec3f  tam   2 2 2
   ]
   {
      Shape {
         appearance Appearance {material MAT_blanco {}}
         geometry Box {size IS tam}
      }
   }
# PROTOTIPO PARED ALUMINIO
PROTO Aluminio [
   field SFVec3f  tam   2 2 2
   ]
   {
      Shape {
         appearance Appearance {material MAT_aluminio {}}
         geometry Box {size IS tam}
      }
   }
# PROTOTIPO CRISTAL
PROTO Cristal [
   field SFVec3f  tam   2 2 2
   ]
   {
      Shape {
         appearance Appearance {material MAT_cristal {}}
         geometry Box {size IS tam}
      }
   }
```

Listado 90: Prototipos de paredes

En el Listado 91 se presenta un ejemplo de cabeceras de prototipos para paredes de distintos colores de la forma en que se deberán definir en cada fichero dónde se los utilice. Nótese que para el prototipo *pared* además de las dimensiones, de especificará la escala (***esc***) y la posible traslación (***tra***) para la textura. Véase el Punto 3.10.3 para más información sobre la palabra reservada **EXTERNPROTO**.

```
#VRML V2.0 utf8

EXTERNPROTO Blanca [field SFVec3f tam] "prototipos.wrl#Blanca"
EXTERNPROTO Aluminio [field SFVec3f tam] "prototipos.wrl#Aluminio"
EXTERNPROTO Cristal [field SFVec3f tam] "prototipos.wrl#Cristal"

EXTERNPROTO Pared [field SFVec3f tam field SFVec2f esc field SFVec2f tra]
"prototipos.wrl#Pared"
```

Listado 91: Cabeceras de prototipos para paredes

El Listado 92 presenta el código LEX para la primera pasada de optimización que dará lugar al programa *primera*.

```
%{
/* EL LENGUAJE VRML97
 * DANIEL HECTOR STOLFI ROSSO - 2004-2009
 *
 * programa que reemplaza los prototipos de paredes
 * y además elimina los comentarios del fichero
 *
 */
%}
%START EXTPROTO PROTO PARED ESC TRA
%%
"#"|"#".+            {;}
<EXTPROTO>.+         {ECHO; BEGIN 0;}
<PROTO>[0-9. ]+\}    {printf (" geometry Box {size%s}",yytext);
                      BEGIN 0;}
<PARED>[0-9. ]+esc   {yytext[yyleng-3]=0;
                      printf (" geometry Box {size%s}\nappearance
                      Appearance {material MAT_opaco{}\ntexture
                      ImageTexture {url \"pared_ladrillos.gif\"}
                      \ntextureTransform TextureTransform {scale",yytext);
                      BEGIN ESC;}
<ESC>[0-9. ]+tra     {printf ("%s",yytext);
                      BEGIN TRA;}
<ESC>[0-9. ]+\}      {printf ("%s}}",yytext);
                      BEGIN 0;}
<TRA>[0-9. -]+\}     {printf ("nslation%s}}",yytext);
                      BEGIN 0;}
"EXTERNPROTO "       {ECHO; BEGIN EXTPROTO;}
"Blanca {tam"        {printf ("Shape {appearance Appearance {material
                      MAT_blanco {}}"); BEGIN PROTO;}
"Aluminio {tam"      {printf ("Shape {appearance Appearance {material
                      MAT_aluminio {}}"); BEGIN PROTO;}
"Cristal {tam"       {printf ("Shape {appearance Appearance {material
                      MAT_cristal {}}"); BEGIN PROTO;}
"Pared {tam"         {printf ("Shape {"); BEGIN PARED;}
.                    {ECHO;}
%%
void main(){
   yylex();
}
```

Listado 92: Código fuente LEX para la primer pasada de optimización

Nótese que el programa se encarga de descartar todos los comentarios, luego elimina las cabeceras en donde se declaran los prototipos a través de la palabra reservada **EXTERNPROTO** que posteriormente se iban a utilizar dentro del fichero.

A continuación, al encontrar una referencia a un prototipo si se tratan de paredes lisas (prototipos: Blanca, Aluminio y Cristal), se reemplazan por el código correspondiente a la pared modelada con ese material el cual también es un prototipo cuyo nombre comienza con las cuatro letras "MAT_"

(convención tomada para este ejemplo) para procesarlo en la segunda pasada y acto seguido se pasa al estado léxico **<PROTO>,** en donde se terminará de escribir el código correspondiente a la pared, que será común para todas ellas.

Si, en cambio, se trata de una pared con textura de ladrillos el estado léxico al que se cambia es el llamado **<PARED>** en donde se definen las dimensiones (campo **tam**) para la caja que hará las veces de pared y se continúa analizando la escala de la textura (campo **esc**) en el estado léxico **<ESC>**. En caso de que con el prototipo también se hubiera utilizado el desplazamiento de textura (campo **tra**) se pasará al estado léxico **<TRA>** el cual terminará de escribir el código para el nodo.

En el Listado 93 se presenta el código fuente de un ejemplo para ser optimizado.

```
#VRML V2.0 utf8

EXTERNPROTO MAT_negro [] "materiales.wrl#MAT_negro"
EXTERNPROTO Pared [field SFVec3f tam field SFVec2f esc field SFVec2f tra]
"prototipos.wrl#Pared"
EXTERNPROTO Blanca [field SFVec3f tam] "prototipos.wrl#Blanca"

# PARED COMPLETA
Pared {tam 1 2 3 esc 5 2 tra .1 .15}

# PARED SIN TRANSLACION
Pared {tam 3 2 1 esc 2 3}

# PARED BLANCA
Blanca {tam 1 1 1}

# CILINDRO NEGRO
Shape {
   appearance Appearance {
      material MAT_negro {}
   }
   geometry Cylinder {height 2     radius 1}
}

# CONO SIN PROTOTIPOS
Shape {
   appearance Appearance {
      material Material {diffuseColor .3 .4 .3}
      texture ImageTexture {url "flores.jpg"}
      textureTransform TextureTransform {scale 4 5}
   }
   geometry Cone {height .4 bottomRadius .15}
}

# ESFERA SIN PROTOTIPOS
Shape {
   appearance Appearance {
      material Material {diffuseColor 1 1 1}
      texture ImageTexture {url "flores.jpg"}
      textureTransform TextureTransform {scale 2 2}
   }
   geometry Sphere {radius 1}
}
```

Listado 93: Fichero de ejemplo a ser optimizado

En el ejemplo de código que se ha escogido se utiliza un prototipo de material (MAT_negro), dos de paredes (Pared y Blanca) y a continuación se modela un cono y una esfera con la misma textura

(*flores.jpg*) la cual es distinta a la que utiliza el prototipo Pared (*pared_ladrillos.gif*) con el fin de que existan dos ficheros de texturas diferentes dentro del código. De esta manera se comprobará más adelante, el funcionamiento de la tercera pasada de optimización en el Punto 4.4.5.

El programa será llamado desde la línea de comandos con la siguiente orden:

```
primera < fichero.wrl > temp.dat
```

Nótese que se dirige a la entrada estándar del programa *primera* el contenido del fichero a optimizar llamado *fichero.wrl* y la salida del programa, correspondiente a la primera optimización, se almacenará en el fichero *temp.dat*. La utilización de este fichero temporal es necesaria al no poder utilizar como entrada y como salida al mismo fichero.

En el Listado 94 se observa el código obtenido luego de la primera pasada de optimización (fichero *temp.dat*). Se puede observar la aparición de líneas en blanco, correspondiéndose con los comentarios y cabeceras eliminadas y el reemplazo de los prototipos de las paredes por el correspondiente código VRML.

```
EXTERNPROTO MAT_negro [] "materiales.wrl#MAT_negro"
EXTERNPROTO Pared [field SFVec3f tam field SFVec2f esc field SFVec2f tra]
"prototipos.wrl#Pared"
EXTERNPROTO Blanca [field SFVec3f tam] "prototipos.wrl#Blanca"


Shape { geometry Box {size 1 2 3 }
appearance Appearance {material MAT_opaco{}
texture ImageTexture {url "pared_ladrillos.gif"}
textureTransform TextureTransform {scale 5 2 translation .1 .15}}}


Shape { geometry Box {size 3 2 1 }
appearance Appearance {material MAT_opaco{}
texture ImageTexture {url "pared_ladrillos.gif"}
textureTransform TextureTransform {scale 2 3}}}


Shape {appearance Appearance {material MAT_blanco {}} geometry Box {size 1 1 1}}


Shape {
   appearance Appearance {
      material MAT_negro {}
   }
   geometry Cylinder {height 2     radius 1}
}


Shape {
   appearance Appearance {
      material Material {diffuseColor .3 .4 .3}
      texture ImageTexture {url "flores.jpg"}
      textureTransform TextureTransform {scale 4 5}
   }
   geometry Cone {height .4 bottomRadius .15}
}


Shape {
```

```
   appearance Appearance {
      material Material {diffuseColor 1 1 1}
      texture ImageTexture {url "flores.jpg"}
      textureTransform TextureTransform {scale 2 2}
   }
   geometry Sphere {radius 1}
}
```

Listado 94: Fichero de ejemplo luego de la primera pasada de optimización

4.4.4 Segunda pasada

En la segunda pasada se van a reemplazar todos los prototipos de materiales: Los que estaban en un principio en el fichero original y los que se han añadido una vez realizada la primera pasada.

Para llevar a cabo este cometido se ha escrito el código LEX del Listado 96 con el cual se traducirán los prototipos de materiales al código VRML correspondiente, para ello es necesaria una estructura de datos auxiliar contenida en el fichero *tab_mat.c*, la que se presenta a continuación en el Listado 95. El resto del código se lo puede consultar en el Apéndice A, junto con los otros listados en C.

```
#define NODO struct reg
struct reg {
   char material[32];
   char descrip[128];
   NODO* sig;
};
```

Listado 95: Estructura utilizada dentro del fichero tab_mat.c

La utilidad de la estructura y de sus funciones asociadas consiste en guardar los registros correspondientes a la identificación del material junto con su definición en código VRML, para su posterior búsqueda y reemplazo dentro del fichero a procesar. Antes de utilizarla se deberá llamar desde el código de *segunda* a la función **inicializa** la que recorrerá el fichero *mat.dat* explicado anteriormente cargando los datos dentro de la estructura.

```
#include "tab_mat.c"
%{
/* EL LENGUAJE VRML97
 * DANIEL HECTOR STOLFI ROSSO - 2004-2009
 *
 * programa que reemplaza los prototipos de materiales
 *
 */
   NODO* t;
%}
%START MATE PROTO
%%
"EXTERNPROTO"     {BEGIN PROTO;}
<PROTO>.+         {BEGIN 0;}
"material MAT_"   {BEGIN MATE;}
<MATE>.+\{\}      {char buf[128];
                   strcpy(buf,"MAT_");
                   strcat(buf,yytext);
                   buf[yyleng+2]=0;
                   if (buf[yyleng+1]==' ') buf[yyleng+1]=0;
                   if (buscar(t,buf)){
                      printf("material %s",buf);
                   } else {
                      printf ("ERROR: No encontrado %s",buf);
```

```
                    }
                    BEGIN 0;}
.                  {ECHO;}
%%
void main(){
   if (inicializa(&t)){
      exit(-1);
   }
   yylex();
}
```

Listado 96: Código fuente LEX para la segunda pasada de optimización

Esta pasada consiste en el análisis lexicográfico del fichero a optimizar, eliminando las cabeceras para los prototipos de los materiales y reemplazando cada ocurrencia de utilización de los mismos por el código extraído de la estructura de apoyo mediante la función **buscar**. Esta función devuelve "**1"** si el material ha sido encontrado disponiéndose en ***buf*** de la cadena a insertar en el fichero, mientras que si ha habido un error (se ha utilizado en el fichero un prototipo de material que no estaba definido en *materiales.wrl*), **buscar** devolverá **"0"** y se insertará la cadena con el error dentro del fichero que se está procesando. Posteriormente cuando este fichero sea leído por el intérprete VRML producirá un error, imprimiéndose en la consola el código inválido encontrado que consistirá en el mensaje de error que insertó esta pasada de optimización.

El código VRML del fichero de ejemplo, una vez realizada la segunda pasada de optimización, es el que se presenta en el Listado 97. Nótese la ausencia de cabeceras de declaración y que los prototipos de los materiales han sido reemplazados por su definición.

```
Shape { geometry Box {size 1 2 3 }
appearance Appearance {material Material {ambientIntensity .6}
texture ImageTexture {url "pared_ladrillos.gif"}
textureTransform TextureTransform {scale 5 2 translation .1 .15}}}


Shape { geometry Box {size 3 2 1 }
appearance Appearance {material Material {ambientIntensity .6}
texture ImageTexture {url "pared_ladrillos.gif"}
textureTransform TextureTransform {scale 2 3}}}


Shape {appearance Appearance {material Material {ambientIntensity .5 diffuseColor 1 1 1}}
geometry Box {size 1 1 1}}


Shape {
   appearance Appearance {
      material Material {ambientIntensity .6 diffuseColor 0 0 0}
   }
   geometry Cylinder {height 2     radius 1}
}


Shape {
   appearance Appearance {
      material Material {diffuseColor .3 .4 .3}
      texture ImageTexture {url "flores.jpg"}
      textureTransform TextureTransform {scale 4 5}
   }
   geometry Cone {height .4 bottomRadius .15}
```

```
}


Shape {
   appearance Appearance {
      material Material {diffuseColor 1 1 1}
      texture ImageTexture {url "flores.jpg"}
      textureTransform TextureTransform {scale 2 2}
   }
   geometry Sphere {radius 1}
}
```

Listado 97: Fichero de ejemplo luego de la segunda pasada de optimización

La llamada desde la línea de comandos al programa *segunda* se hará en conjunción con la tercera pasada (programa *tercera*) utilizando una tubería para evitar la necesidad de otro fichero temporal.

4.4.5 Tercera pasada

Para la tercera pasada de optimización, se ha hecho algo similar a lo realizado con los materiales en la segunda, pero en este caso con las texturas. El objetivo consiste en evitar utilizar *urls* referenciando al mismo fichero de textura repetidas veces, utilizando en su lugar referencias a una primera y única *url* para cada textura diferente dentro de cada fichero. Además al ser esta la última pasada ha sido necesario restaurar la cabecera VRML para el fichero que fue suprimida junto con los comentarios por la primera pasada y eliminar los tabuladores, espacios extras y retornos de carro en detrimento del aspecto legible pero en búsqueda de un tamaño mínimo de fichero.

Se ha utilizado como estructura auxiliar para esta pasada, otra lista enlazada definida en el fichero *tab_tex.c* (véase el código completo del fichero en el Apéndice A y la porción correspondiente a la definición de la estructura en el Listado 98). De esta manera, al recorrer el fichero a optimizar se irán guardando las primeras ocurrencias de cada textura agregando en el código VRML una referencia por medio de la cadena **DEF** TEXnn, donde TEX es una convención arbitraria elegida para este ejemplo, nn es el número de orden con el cual se identificará de ahora en más a esta textura y **DEF** es la palabra reservada por VRML para crear referencias como ya se ha visto en el Punto 3.10.1.

Cuando se vuelva a utilizar la misma textura dentro del código fuente, al encontrarse ya en la lista enlazada, será reemplazada en el código de salida por una referencia a la definida previamente utilizando la cadena **USE** TEXnn, donde el número nn se obtendrá del guardado en la estructura auxiliar.

```
#define NODO struct reg
struct reg {
   int indice;
   char descrip[128];
   NODO* sig;
};
```

Listado 98: Estructura utilizada dentro del fichero tab_tex.c

En el Listado 99 se observa el código LEX para la tercera pasada. Nótese el empleo de la función **buscar** que, en este caso si no encuentra la textura buscada devuelve el valor de índice **"0"**, creándose la primer referencia a la textura mediante la palabra reservada **DEF** e insertando (función **insertar**) la cadena con el nombre del fichero y el nuevo índice asignado en la estructura auxiliar; mientras que si se la encuentra, el valor devuelto por **buscar** será el índice con el cual construir la copia por referencia junto con la cadena "TEX" y utilizando la palabra reservada **USE**.

```
#include "tab_tex.c"
%{
/* EL LENGUAJE VRML97
 * DANIEL HECTOR STOLFI ROSSO - 2004-2009
 *
 * Programa que toma un textura y reemplaza sus usos siguientes
 * por una referencia a la misma
 * Ademas reestablece la cabecera del fichero VRML y transforma
 * todas las extensiones .wrl por .wrz
 *
 */
   NODO* t;
   int cont;
%}
%START TEX
%%
"ImageTexture {url "       {BEGIN TEX;}
<TEX>" "                   {;}
<TEX>\"[a-zA-Z0-9._]+\"\}  {char buf[128];
                            int num;
                            if (num = buscar(t,yytext)){
                              printf("USE TEX%d",num);
                            } else {
                              sprintf(&buf,"TEX%d",cont);
                              printf("DEF %s ImageTexture {url %s",buf,yytext);
                              insertar(&t,cont++,yytext);
                            }
                            BEGIN 0;}
".wrl"                     {printf(".wrz");}
[\t|\n| ]+                 {printf(" "); }
.                          {ECHO;}
%%
void main(){
   cont = 1;
   printf("#VRML V2.0 utf8\n");
   yylex();
}
```

Listado 99: Código fuente LEX para la tercera pasada de optimización

Además esta pasada de optimización reemplazará todas las extensiones en las *urls* de los ficheros *.WRL* por *.WRZ*. Esto se corresponde con la última optimización a realizar en aras de minimizar el tamaño del fichero y que será explicada en el Punto 4.4.6.

Por último se sustituirán todas las sucesiones de tabuladores, espacios y retornos de carro por un solo espacio, ya que la función de estos símbolos en VRML es puramente decorativa persiguiendo fines de claridad en el código, pero nuestro objetivo aquí es mejorar los tiempos de descarga minimizando el tamaño del fichero.

Al finalizar esta tercera pasada se tendrá un fichero con la cabecera VRML, que fue suprimida por la primera junto con los comentarios y que será añadida ahora por la función **main** (véase el código en el Listado 99), seguida por una sola línea de código con el resto del contenido del fichero *.WRL*.

Como se mencionó al finalizar el punto anterior la llamada desde la línea de comandos en este caso lanzará a las dos pasadas conjuntamente. De esta manera se evita el uso de un segundo fichero temporal haciendo uso en su lugar de una tubería:

```
segunda < temp.dat | tercera > fichero.wrl
```

El contenido del fichero del ejemplo, totalmente optimizado a través de las tres pasadas descritas, se puede observar en el Listado 100. Como se desprende del mismo se ha perdido toda legibilidad pero se ha ganado mucho en eficiencia.

```
#VRML V2.0 utf8
 Shape { geometry Box {size 1 2 3 } appearance Appearance {material Material
{ambientIntensity .6} texture DEF TEX1 ImageTexture {url "pared_ladrillos.gif"}
textureTransform TextureTransform {scale 5 2 translation .1 .15}}} Shape { geometry Box
{size 3 2 1 } appearance Appearance {material Material {ambientIntensity .6} texture USE
TEX1 textureTransform TextureTransform {scale 2 3}}} Shape {appearance Appearance
{material Material {ambientIntensity .5 diffuseColor 1 1 1}} geometry Box {size 1 1 1}}
Shape { appearance Appearance { material Material {ambientIntensity .6 diffuseColor 0 0
0} } geometry Cylinder {height 2 radius 1} } Shape { appearance Appearance { material
Material {diffuseColor .3 .4 .3} texture DEF TEX2 ImageTexture {url "flores.jpg"}
textureTransform TextureTransform {scale 4 5} } geometry Cone {height .4 bottomRadius .
15} } Shape { appearance Appearance { material Material {diffuseColor 1 1 1} texture USE
TEX2 textureTransform TextureTransform {scale 2 2} } geometry Sphere {radius 1} }
```

Listado 100: Ejemplo luego de la tercera pasada de optimización

Además de eliminar el uso de prototipos y al utilizar referencias a texturas, han disminuido los recursos locales que se ocupan y, si bien a crecido levemente el tamaño del fichero en Bytes (ha pasado de 990 a 1034) esto no es importante porque todavía queda por realizar la última optimización con la cual se solucionará de forma radical el problema del tamaño del fichero a descargar desde Internet.

4.4.6 Compresión

Para abordar esta última fase de optimización, se hará uso de una utilidad de compresión externa: El programa *GZIP*[32]

Aprovechando que los *plug-ins* actuales permiten las transferencias de ficheros VRML comprimidos se disminuirá considerablemente la cantidad de Bytes a transferir que luego serán descomprimidos localmente. La convención recomendada por los *plug-ins* es la de utilizar la extensión *.WRZ* en lugar de *.WRL.GZ* (la que genera por defecto *GZIP*) permitiendo diferenciar fácilmente un fichero *.WRL* comprimido de cualquier otro fichero *.GZ*. Por estas razones será también la extensión que se adopte en este ejemplo.

[32] GNU ZIP: Utilidad de compresión, que utiliza un algoritmo libre de patentes, escrita por Jean-loup Gailly y Mark Adler

El proceso de compresión de cada fichero se llevará a cabo luego de la tercer pasada de optimización, por lo que esta instrucción también se ejecutará desde el fichero *script.bat* el cual, recuérdese, es construido de forma automática por el programa *control*.

De esta manera para nuestro fichero de ejemplo, el fichero *script.bat* contendrá las instrucciones que se observan en el Listado 101.

```
primera < fichero.wrl > temp.dat
segunda < temp.dat | tercera > fichero.wrl
gzip -9 fichero.wrl
move /y fichero.wrl.gz fichero.wrz
```

Listado 101: Fichero script.bat generado de forma automática

Nótese que las dos primeras líneas son las que se han expuesto con anterioridad para las tres primeras pasadas de optimización, mientras que la tercera responde a la invocación del programa *GZIP*, indicando la máxima compresión (**-9**) seguido del nombre del fichero a comprimir. Por último, en la cuarta línea se cambia la extensión por defecto (*WRL.GZ*) por la recomendada para este caso (*.WRZ*).

Obsérvese que en vez del comando **ren** se ha utilizado para el renombrado el comando **move**, esto responde a la ventaja de poder emplear para este último el modificador (**/Y**) permitiendo sobrescribir al fichero destino si éste ya existía. Así no será necesario borrar manualmente los ficheros antiguos al realizar una actualización.

De esta manera el fichero del ejemplo que originalmente ocupaba 990 Bytes, tras las tres pasadas y la posterior compresión, pasó a ocupar 352 Bytes.

4.5 Resumen

En este punto se abordaron las optimizaciones en búsqueda de un modelo que se comporte de manera más fluida, evitando sobrecargas computacionales innecesarias y también en aras de reducir el tamaño del modelo para que sea transferido de forma más eficiente por Internet.

Debido a que con los años irá mejorando la informática actual, ganando potencia de proceso y a que Internet será cada día más veloz, quizás no sea necesario poner tanto empeño en estas optimizaciones aunque siempre la eficiencia será bienvenida como un buen hábito de programación.

Capítulo 5

Ejemplos

5.1 Tapiz de proyección

Este ejemplo consiste en un tapiz para un proyector que simula el desenrollado y enrollado del mismo al actuar sobre su cordel. En la Figura 63 se observa el diagrama en bloques que define el funcionamiento del *script*. Básicamente consiste en escalar y desplazar la primitiva utilizada para el tapiz simulando la recogida del mismo.

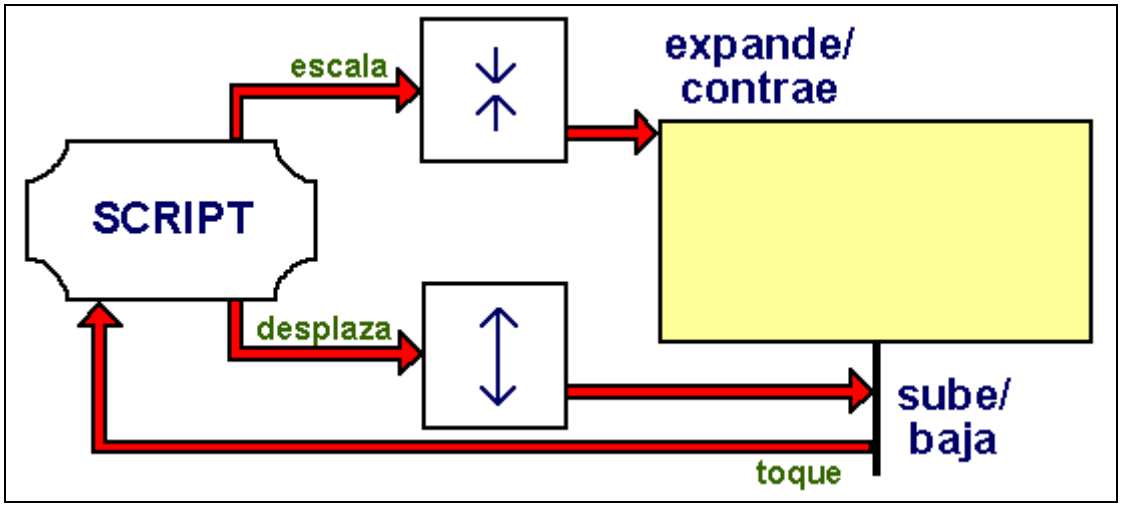


Figura 63: Eventos del script para el tapiz de proyección

En el código del *script* (Listado 102) se puede analizar el código **vrmlscript** que está compuesto por la función ***toque*** que se llama cada vez que se activa el **TouchSensor *TS***. Esta función según el estado de la variable ***bajado*** escalará y desplazará el tapiz para expandirlo si se encontraba recogido o para comprimirlo si estaba desplegado.

Nótese que a través de los eventos de salida ***escala*** y ***desplaz*** se controla el tamaño y la posición de la primitiva del tapiz.

```
#VRML V2.0 utf8
########################################
# Tapiz de proyección                  #
#                                      #
# EL LENGUAJE VRML97                   #
# DANIEL HECTOR STOLFI ROSSO – 2004-2009 #
########################################

PROTO MAT_tapiz [] {Material {ambientIntensity .7 diffuseColor .9 .9 .8}}
PROTO MAT_aluminio [] {Material {diffuseColor .55 .55 .55 ambientIntensity .5}}
```

```
PROTO MAT_cordel [] {Material {diffuseColor 1 1 0}}

DEF TAPIZ Transform {
   translation 0 -.95 -.01
   scale 1 .15 1
   center 0 .95 0
   children Shape {
         appearance Appearance {material MAT_tapiz {}}
         geometry Box {size 2.4 1.9 .01}
   }
}
DEF CORDEL Transform {
   translation 0 -.53 -.01
   children [
      Shape {
         appearance Appearance {material MAT_cordel {}}
         geometry Cylinder {
            radius 0.01
            height 0.5
         }
      }
      DEF TS TouchSensor { }
   ]
}
Transform {
   translation 0 0 0
   rotation 0 0 1 1.5707
   children Shape {
      appearance Appearance {material MAT_aluminio {}}
      geometry Cylinder {height 2.5 radius .05 }
   }
}
Transform {
   translation 0 0 -.03
   rotation 0 0 1 1.5707
   children Shape {
      appearance Appearance {material MAT_aluminio {}}
      geometry Cylinder {height 6 radius .02 }
   }
}
DEF SC Script {
   eventIn  SFTime   toque
   eventOut SFVec3f  escala
   eventOut SFVec3f  desplaz
   field    SFBool   bajado   FALSE
   url "vrmlscript:
      function toque() {
         if (bajado) {
            bajado = FALSE;
            escala = new SFVec3f(1,.15,1);
            desplaz = new SFVec3f(0,-.53,-.01);
         }
         else {
            bajado = TRUE;
            escala = new SFVec3f(1,1,1);
            desplaz = new SFVec3f(0,-2.15,-.01);
         }
      }
   "
}
ROUTE TS.touchTime TO SC.toque
ROUTE SC.escala TO TAPIZ.set_scale
ROUTE SC.desplaz TO CORDEL.set_translation
```

Listado 102: Código fuente del tapiz de proyección.

A continuación, en la Figura 64, se observa el tapiz del ejemplo desplegado.

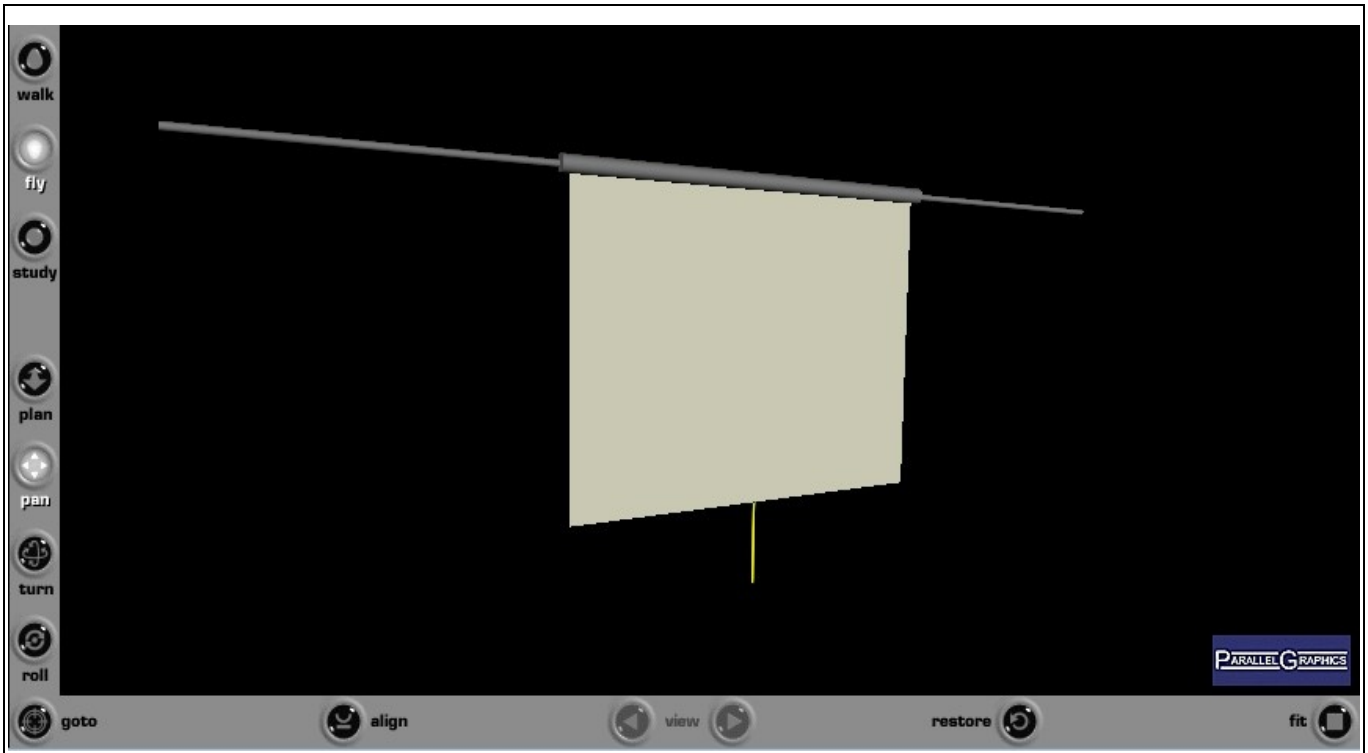


Figura 64: Ejemplo del tapiz de proyección

5.2 Barrera

El funcionamiento de la barrera es similar al del Tapiz de proyección en su funcionamiento pero en este caso se ha evitado la utilización de un *script* para que realice la animación ya que sólo basta con las primitivas que provee VRML. En la Figura 65 se puede contemplar los eventos que se disparan al pulsar el botón y al variar los valores del nodo rampa.

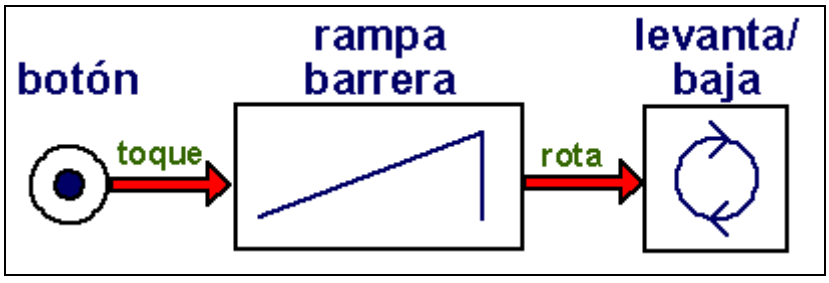


Figura 65: Eventos del script para la barrera

Nótese en el Listado 103 el enrutado de eventos entre el **TouchSensor** ***TS*** y el **TimeSensor** ***rampa*** y entre ésta y el **OrientationInterpolator** ***Abre_cierra*** y entre éste último y el conjunto de nodos ***barrera*** afectando la rotación del mismo.

```
#VRML V2.0 utf8
##########################################
# Barrera                                #
#                                        #
# EL LENGUAJE VRML97                     #
# DANIEL HECTOR STOLFI ROSSO – 2004-2009 #
```

```
########################################

PROTO MAT_barrera [] {Material {ambientIntensity .6}}
PROTO MAT_boton [] {Material {ambientIntensity .6 emissiveColor 0 1 0}}
PROTO MAT_rojo [] {Material {ambientIntensity .6 diffuseColor .796 .254 .254}}

Transform {
   translation -5 -1 0
   children Shape {
      appearance Appearance {material MAT_barrera {}}
      geometry Box {size .2 1 .2}
   }
}
Transform {
   translation -4.9 -0.6 0
   rotation 0 0 1 1.5708
   children Group {
      children [
         Shape {
            appearance Appearance {material MAT_boton {}}
            geometry Cylinder {height .05 radius .05}
         }
         DEF TS TouchSensor { }
      ]
   }
}
Transform {
   translation -5.5 -0.85 -5
   children Shape {
      appearance Appearance {material MAT_rojo {}}
      geometry Box {size .4 1.3 .4}
   }
}
Transform {
   translation 4 -0.85 -5.15
   children Shape {
      appearance Appearance {material MAT_rojo {}}
      geometry Box {size .1 1.3 .1}
   }
}
DEF barrera Transform {
   translation -0.75 -0.4 -5.25
   center -4.75 0 0
   rotation 0 0 1 0
   children
   Group {
      children [
         Transform {
            translation .15 0 0
            children Shape {
               appearance Appearance {material MAT_barrera {}}
               geometry Box {size 10.5 .2 .1}
             }
         }
         Transform {
            translation -5.3 0 0
            children Shape {
               appearance Appearance {material MAT_rojo {}}
               geometry Box {size .4 .4 .1}
             }
         }
      ]
   }
}
DEF Rampa TimeSensor {
   cycleInterval 10
   loop  FALSE
}
DEF Abre_Cierra OrientationInterpolator {
   key      [0,   .1,       .9,     1]
   keyValue [0 0 1 0, 0 0 1 1.4, 0 0 1 1.4, 0 0 1 0]
}
```

```
ROUTE TS.touchTime TO Rampa.startTime
ROUTE Rampa.fraction_changed TO Abre_Cierra.set_fraction
ROUTE Abre_Cierra.value_changed TO barrera.set_rotation
```

Listado 103: Código fuente de la barrera

En la Figura 66 se puede contemplar el ejemplo de la barrera ejecutándose en la ventana del *plug-in*.

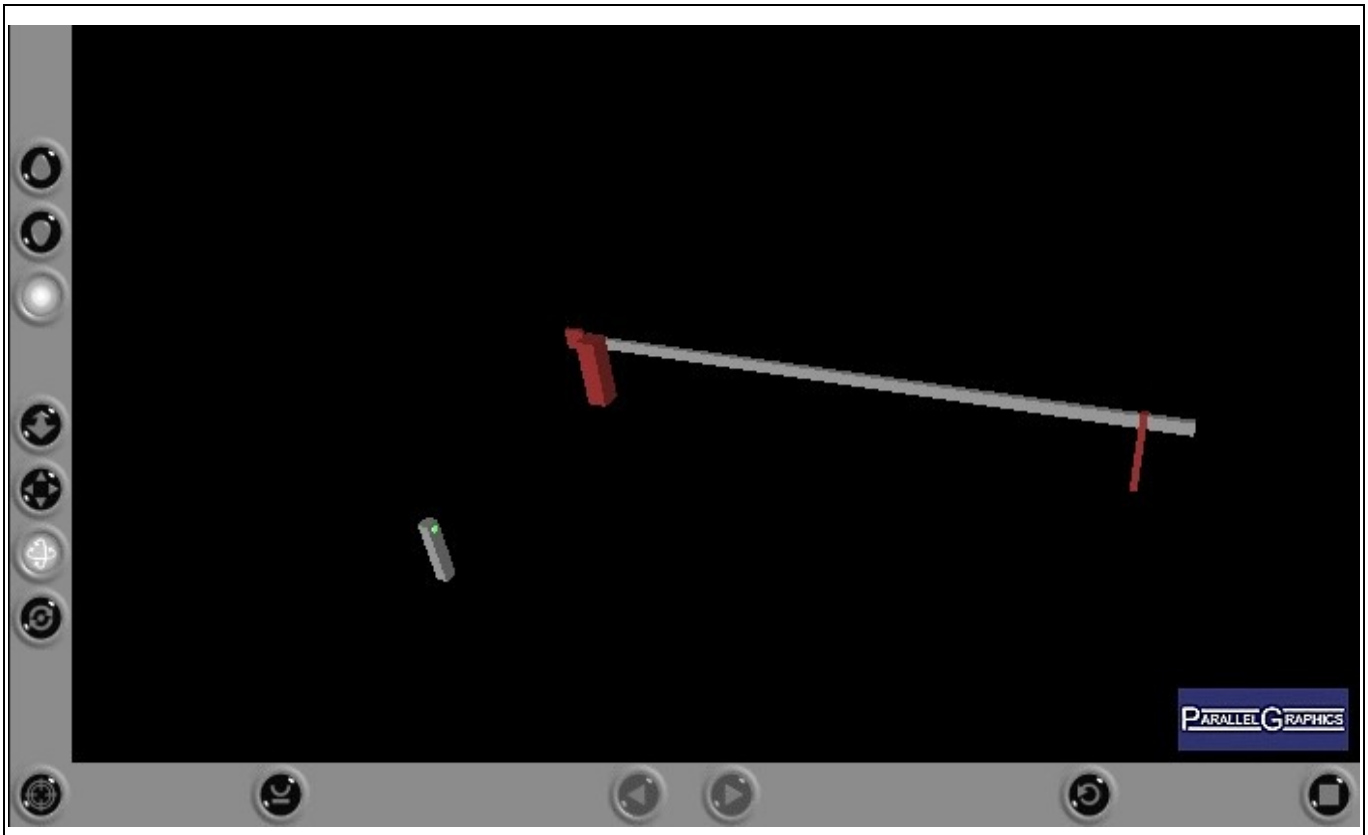


Figura 66: Ejemplo de la barrera

5.3 Pizarra virtual

Este ejemplo contiene un *script* más complejo en el cual se simula el comportamiento de una pizarra con tres tizas de colores y un borrador.

Primero se definen las condiciones del entorno, color de fondo, ubicación inicial de la cámara, etc. Posteriormente le toca el turno a las primitivas, comenzando con la tiza (***TR***), el material con el que está construida (***MAT***), la pizarra que contiene un **TouchSensor (*TS*)**, el grupo de nodos hijos de la pizarra **(*GR*)** inicialmente vacío, el borrador **(*BO*)**, la tiza roja **(*RO*)**, la tiza blanca **(*BL*)** y la tiza azul **(*AZ*)** todos ellos con su correspondiente **TouchSensor** asociado. Por último se define es *script* y el entrutado de los eventos.

El comportamiento del mundo virtual es el siguiente. Inicialmente se visualiza la pizarra con tres tizas de colores y el borrador. Pulsando sobre una de las tizas se dispara el evento del correspondiente **TouchSensor** (***RO***, ***BL*** o ***AZ***) que se enruta al *script* ***SC*** asociándolo al evento correspondiente (***rojo***,

blanco ó ***azul***). Estos eventos se encargarán de coger o dejar la tiza dependiendo de si se la tenía en la mano (movimiento solidario con el puntero) o no.

Al coger una tiza también se habilita el **TouchSensor *TS*** en la pizarra a través del evento de salida ***puntero*** del *script*, ahora al mover sobre la pizarra el puntero se moverá la tiza (evento ***TS.hitPoint_changed*** enrutado a ***TR.set_translation***) y si se mantiene activo se dibujará un trazo en la pantalla que no es más que nuevos puntos que se van creando como hijos de la pizarra (grupo ***GR***) desde el *script* mediante los eventos recibidos desde el **TouchSensor *TS*** por el evento de entrada ***pinta***. La forma de añadir puntos es pasando la cadena de código que los define al evento ***GR.addChildren***. La pulsación sobre la pizarra se indica dentro del *script* mediante la variable de estado ***pulsa*** la que se activa y desactiva desde el evento de entrada ***tiza***.

Para borrar la pizarra basta con pulsar sobre el borrador el cual dispara el evento ***borra*** dentro del *script* asignando al grupo de nodos hijos de la pizarra el conjunto vacío.

```
#VRML V2.0 utf8
########################################
# Pizarra Virtual                      #
#                                      #
# EL LENGUAJE VRML97                   #
# DANIEL HECTOR STOLFI ROSSO – 2004-2009 #
########################################

NavigationInfo {
   headlight FALSE
}
DirectionalLight {
   direction -1 -1 -1
   ambientIntensity 1
   intensity .7
}
Background {
   skyAngle [0, 3.14]
   skyColor [.7 .7 .7]
}
Viewpoint {
   position 6 0.5 3
}
Transform {
   translation 5.25 -.025 .05
   children
   Shape {
      appearance Appearance {
         material Material {
            diffuseColor .55 .55 .55
            ambientIntensity .5
         }
      }
      geometry Box {size 10.5 .05 .1}
   }
}
# TIZA EN MOVIMIENTO
DEF TR Transform {
   translation 7.03 .02 0
   children
   Transform {
      translation .07 0 .05
      rotation 0 1 0 .7
```

```
      children
      Shape {
         appearance Appearance {
            material DEF MAT Material {
               diffuseColor 1 1 1
               ambientIntensity 1
            }
         }
         geometry Box {size .02 .02 .1}
      }
   }
}
# GRUPO DINAMICO: PIZARRA

Group {
   children [
      Shape {
         appearance Appearance {
            material Material {
               ambientIntensity .8
               diffuseColor 0 .4 0
            }
         }
         geometry IndexedFaceSet {
            coord Coordinate {point [0 0 0, 10.5 0 0, 10.5 1.25 0, 0 1.25 0]}
            coordIndex [0 1 2 3 -1 ]
            solid FALSE
         }
      }
      DEF TS TouchSensor {enabled FALSE }
      DEF GR Group { }
   ]
}
# BORRADOR
Transform {
   translation 7.6 .02 .04
   children [
      DEF BO TouchSensor { }
      Shape {
         appearance Appearance {
            material Material {
               ambientIntensity .5
               diffuseColor .910 .867 .731
            }
         }
         geometry Box {size .1 .04 .05 }
      }
      Transform {
         translation 0 .025 0
         children
            Shape {
               appearance Appearance {
                  material Material {
                     ambientIntensity .5
                     diffuseColor .324 .16 0
                  }
               }
               geometry Box {size .1 .01 .05}
            }
       }
   ]
}
# TIZA ROJA
Transform {
   translation 7 .02 .05
   rotation 0 1 0 0.7
   children [
      DEF RO TouchSensor { }
      Shape {
         appearance Appearance {
            material Material {
               diffuseColor 1 0 0
```

```
               ambientIntensity 1
            }
         }
         geometry Box {size .02 .02 .1}
      }
   ]
}
# TIZA BLANCA
Transform {
   translation 7.1 .02 .05
   rotation 0 1 0 0.7
   children [
      DEF BL TouchSensor { }
      Shape {
         appearance Appearance {
            material Material {
               diffuseColor 1 1 1
               ambientIntensity 1
            }
         }
         geometry Box {size .02 .02 .1}
      }
   ]
}
# TIZA AZUL
Transform {
   translation 7.2 .02 .05
   rotation 0 1 0 0.7
   children [
      DEF AZ TouchSensor { }
      Shape {
         appearance Appearance {
            material Material {
               diffuseColor 0 1 1
               ambientIntensity 1
            }
         }
         geometry Box {size .02 .02 .1}
      }
   ]
}
# SCRIPT
DEF SC Script {
   eventIn  SFVec3f  pinta
   eventIn  SFBool   tiza
   eventIn  SFBool   borra
   eventIn  SFTime   blanco
   eventIn  SFTime   rojo
   eventIn  SFTime   azul
   eventOut MFNode   nodo
   eventOut MFNode   vacia
   eventOut SFColor  colo
   eventOut SFBool   puntero
   eventOut SFVec3f  pos
   field    SFInt32  indice   0
   field    SFBool   pulsa    FALSE

   url "vrmlscript:
      function pinta(valor) {
         if (pulsa) {
            nodo = new MFNode((new SFNode(new String('Shape { geometry PointSet {color
Color {color '+colo[0]+' '+colo[1]+' '+colo[2]+'} coord Coordinate {point [ '+ valor[0]
+' '+ valor[1]+' .015 ]} }}')));
         }
      }
      function tiza(valor) {
         pulsa = valor;
      }
      function borra() {
         vacia = new MFNode();
      }
      function rojo() {
```

```
         if (indice == 1) {
            puntero = FALSE;
            pos = new SFVec3f(7.03,.02,0);
         } else {
            colo = new SFColor(1,0,0);
            puntero = TRUE;
            indice = 1;
         }
         print (puntero);
      }
      function blanco() {
         if (indice == 2) {
            puntero = FALSE;
            pos = new SFVec3f(7.03,.02,0);
         } else {
            colo = new SFColor(1,1,1);
            puntero = TRUE;
            indice = 2;
         }
      }
      function azul() {
         if (indice == 3) {
            puntero = FALSE;
            pos = new SFVec3f(7.03,.02,0);
         } else {
            colo = new SFColor(0,1,1);
            puntero = TRUE;
            indice = 3;
         }
      }
   "
}
ROUTE SC.vacia TO GR.set_children
ROUTE SC.nodo TO GR.addChildren
ROUTE TS.hitPoint_changed TO SC.pinta
ROUTE TS.isActive TO SC.tiza
ROUTE BO.isActive TO SC.borra
ROUTE TS.hitPoint_changed TO TR.set_translation
ROUTE RO.touchTime TO SC.rojo
ROUTE BL.touchTime TO SC.blanco
ROUTE AZ.touchTime TO SC.azul
ROUTE SC.colo TO MAT.set_diffuseColor
ROUTE SC.puntero TO TS.set_enabled
ROUTE SC.pos TO TR.set_translation
```

Listado 104: Código fuente de la pizarra virtual

En la Figura 67 se observa la pizarra virtual tal como se visualiza en la ventana del navegador.

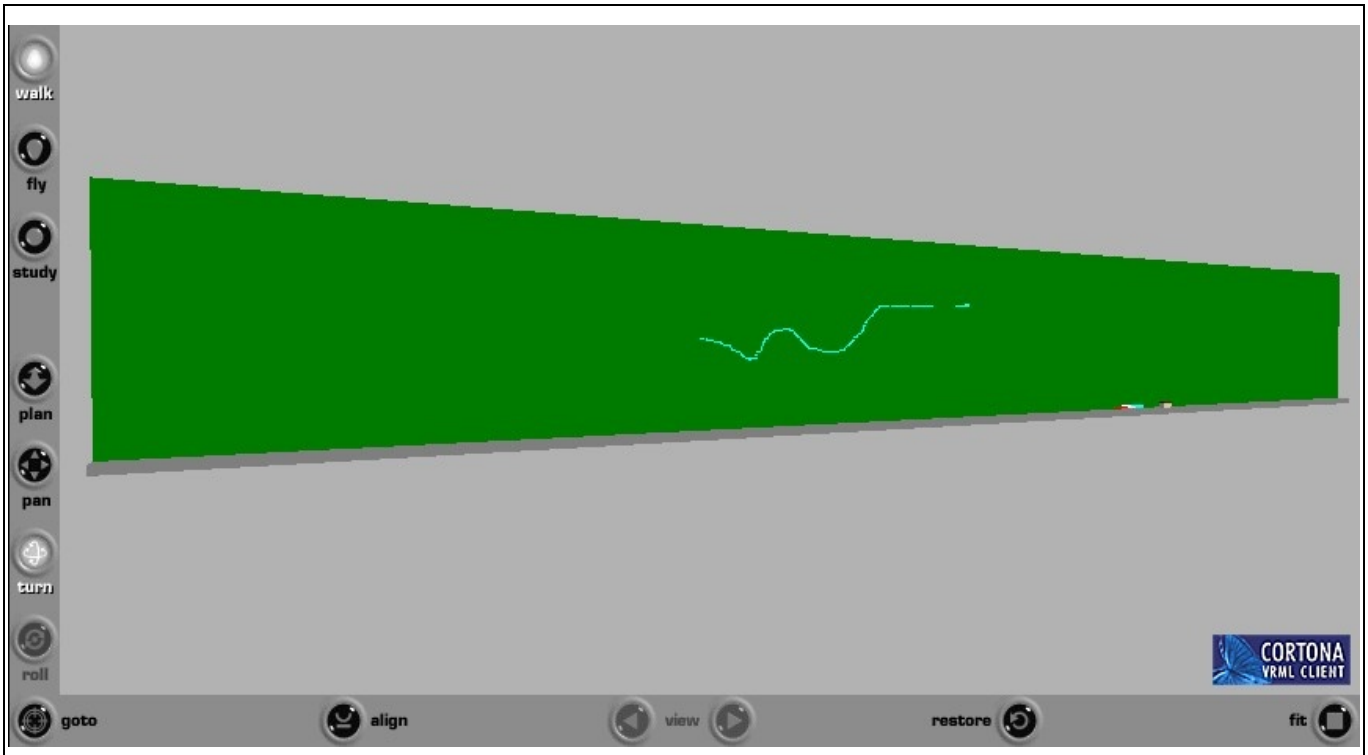


Figura 67: Ejemplo de la pizarra virtual

Capítulo 6

Conclusiones

Hemos comenzado este libro mencionado los orígenes de VRML y la evolución que ha sufrido con el paso del tiempo hasta llegar a la versión actual. Era obligado echar un vistazo a los *plug-ins* más conocidos en el mercado, resaltando las bondades y deficiencias de cada uno. Posteriormente se presentaron las cuatro primitivas principales, para que de este modo se comenzara de una manera sencilla a familiarizarnos los primeros listados de código VRML.

Luego fue el momento de conocer exactamente qué nos permite hacer VRML, para ello se presentó una guía descriptiva sobre los nodos que componen el lenguaje, afrontando ejemplos reales y aportando figuras con los resultados obtenidos siempre que fue posible. Un caso a destacar fue el del nodo **Script**, para el cual se intentó dar una visión general sobre el lenguaje de programación utilizado dentro del mismo.

Por último, conociéndose ya todos los entresijos del lenguaje VRML, se aplicaron técnicas de optimización mediante transformaciones sobre el código, llevadas a cabo por los programas externos escritos para tal fin.

Para escribir un fichero VRML, es más, para desarrollar el código de un mundo virtual completo, no se necesita más que un editor de texto plano. Por supuesto que es de gran ayuda si además el mismo consta de la posibilidad de colorear la sintaxis, colaborando así con la labor de corrección e identificación de nodos, campos y parámetros.

La construcción y tratamiento de las imágenes quedan a cargo de cualquier editor gráfico que sea capaz de manejar los formatos utilizados (principalmente *.GIF* y *.JPG*), y para conseguir que las texturas sean realistas sería de gran utilidad el empleo de una cámara fotográfica digital.

Los sonidos podrán ser obtenidos mediante sintetizadores, o bien grabándolos desde el ambiente real. Luego, será preciso mediante un editor de ondas adecuar las características y el tamaño del fichero.

Y por último, lo primordial: contar con un *plug-in* que permita la visualización de los mundos creados. Si bien, como se ha visto existen extensiones propias que cada fabricante incluye en su producto, no es recomendable la utilización de las mismas debido a la pérdida de compatibilidad con el estándar y la consiguiente limitación de la cantidad de visitantes posibles, quedando el mundo restringido a los que tengan instalado un *plug-in* específico.

Aunque no se ha utilizado ninguna de ellas para los ejemplos de este libro, existen utilidades que permiten modelar primitivas basadas en el nodo **Extrusion**, en el nodo **ElevationGrid**, e incluso para modelos complejos basados en muchos nodos **IndexedFaceSet**. Estos programas serán de gran ayuda en la construcción de formas complejas, pero sólo en estos casos, dado que nunca se alcanzará la misma exactitud que la obtenida en situar, escalar o rotar objetos escribiendo el código fuente de forma manual.

Otros programas se encargan de convertir formatos conocidos de programas CAD/CAM en ficheros VRML, quién escribe estas líneas no ha probado ninguno de ellos pero pueden ser una alternativa al modelado manual dado que estos programas sí constan de una muy buena precisión de diseño.

¿Qué nos depara el futuro? Si nos atenemos a la ley de Moore, aunque carente de rigor, en los próximos cinco años la tecnología del silicio conseguirá llegar hasta 0.03 micras lo que permitirá el aumento de transistores que cabrán en un chip y por ende la potencia de proceso del mismo sumándose a la utilización de una o varias *GPUs*.[33] Internet, intereses económicos mediante, continuará con su progresión aumentando las velocidades de transmisión de datos y evolucionando hacia Internet II, actualmente en desarrollo en universidades y que proporcionará, entre otras cosas, asignación de ancho de banda selectivo según el tipo de contenido que se esté transfiriendo. Aunque, hoy por hoy económicamente inaccesibles al usuario medio, existen en el mercado gafas estereoscópicas y guantes de realidad virtual los que, como ha pasado con otros periféricos, poco a poco irán bajando su precio y haciéndose accesibles a todos los usuarios así como los monitores holográficos hoy en día bajo investigación.

Apoyado en este avance tecnológico, la experiencia de navegar por Internet cambiará inexorablemente hacia un modelo más real e intuitivo y aquí es donde VRML toma relevancia permitiendo visitar mundos interminables y plagados de detalles; comprar en verdaderos supermercados virtuales, caminado entre las góndolas y llenando el carrito de la compra virtual; explorar ambientes inhóspitos como la superficie marciana y, por qué no, asistir a reuniones virtuales junto a contertulios de todo el mundo[34].

En resumen, el futuro son las tres dimensiones porque somos seres tridimensionales, por lo tanto con este libro se ha querido aportar un granito de arena para acercar a nuestro presente, el tan ansiado porvenir.

[33] Graphics Process Unit – Unidad de Proceso Gráfico: Procesador dedicado al renderizado de polígonos y al proceso de texturas, normalmente ubicado en la tarjeta gráfica, es el encargado de acelerar las representaciones tridimensionales.

[34]Nótese la diferencia de lo propuesto frente a programas del tipo SecondLife™ (**http://secondlife.com**) que requieren una aplicación autónoma para ejecutarse y de la cual dependen, mientras que la filosofía expuesta aquí es la universalidad de acceso mediante navegador web y *plug-in* independientemente de la plataforma de ejecución tanto software como hardware.

Apéndice A

Optimizadores en C

```
/* EL LENGUAJE VRML97
 * DANIEL HECTOR STOLFI ROSSO - 2004-2009
 * programa que escribe el script para procesar los ficheros a optimizar
 */
#include <stdio.h>
#include <stdlib.h>
void main() {
   char buf[128];
   char cmd[128];
   while (NULL != (fgets(buf,128,stdin))){
      buf[strlen(buf)-1]=0;
      strcpy(cmd, "primera < ");
      strcat(cmd, buf);
      strcat(cmd, " > temp.dat");
      printf("%s\n",cmd);
      strcpy(cmd, "segunda < temp.dat | tercera > ");
      strcat(cmd, buf);
      printf("%s\n",cmd);
      printf("gzip -9 %s\n",buf);
      printf("move /y %s.gz ",buf);
      buf[strlen(buf)-3]=0;
      printf("%swrz\n",buf);
   }
   exit (0);
}
```

Listado 105: control.c

```
/* EL LENGUAJE VRML97
 * DANIEL HECTOR STOLFI ROSSO - 2004-2009
 * programa que gestiona la lista para almacenar los prototipos de los materiales
 */
#include <stdio.h>
#define NODO struct reg
struct reg {
   char material[32];
   char descrip[128];
   NODO* sig;
};
void insertar (NODO** t,char* nom, char* des){
   NODO* temp;
   temp=(NODO*)malloc(sizeof(NODO));
   if (!temp){
      perror("No se puede asignar memoria\n");
   }
   nom[strlen(nom)-1]=0;
   des[strlen(des)-1]=0;
   strcpy(temp->material,nom);
   strcpy(temp->descrip,des);
   temp->sig=*t;
   *t=temp;
}
int buscar(NODO* t, char* buf) {
   while ((t) && (strcmp(t->material,buf)!=0)) {
      t=t->sig;
   }
   if (t) {
      strcpy(buf,t->descrip);
      return 1;
   }
   return 0;
```

```
}
int inicializa(NODO** t) {
   char buf1[32];
   char buf2[128];
   FILE* fmat;
   if (NULL == (fmat=fopen("mat.dat", "r"))){
      perror("Error abriendo mat.dat\n");
      return 1;
   }
   while (NULL != (fgets(buf1,32,fmat))){
      if (NULL != (fgets(buf2,128,fmat))){
         insertar(t, buf1, buf2);
      }
   }
   fclose(fmat);
   return 0;
}
```

Listado 106: tab_mat.c

```
/* EL LENGUAJE VRML97
 * DANIEL HECTOR STOLFI ROSSO - 2004-2009
 * programa que gestiona la lista para almacenar los nombres de las texturas
 */
#define NODO struct reg
struct reg {
   int indice;
   char descrip[128];
   NODO* sig;
};
void insertar (NODO** t,int num, char* des){
   NODO* temp;
   temp=(NODO*)malloc(sizeof(NODO));
   if (!temp){
      perror("No se puede asignar memoria\n");
   }
   temp->indice=num;
   strcpy(temp->descrip,des);
   temp->sig=*t;
   *t=temp;
}
int buscar(NODO* t, const char* buf) {
   while ((t) && (strcmp(t->descrip,buf)!=0)) {
      t=t->sig;
   }
   if (t) {
      return t->indice;
   }
   return 0;
}
void imprimir(NODO* t){
   printf("\n\nIMPRIMIR:\n");
   while (t){
      printf("%d: %s\n",t->indice,t->descrip);
      t=t->sig;
   }
}
```

Listado 107: tab_tex.c

Apéndice B

Índices

B.1 Índice de Figuras

B.2 Índice de Listados

B.3 Índice de Fórmulas

B.4 Índice de Tablas

Apéndice C

Bibliografía

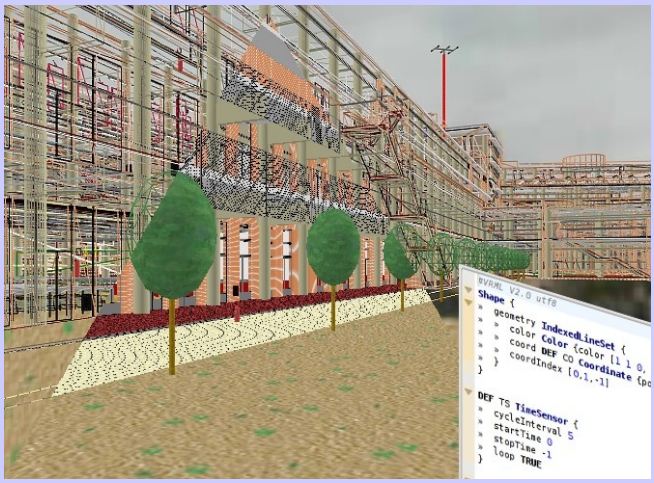
La bibliografía sobre la que se ha apoyado la elaboración de este libro y la cual es de obligada consulta para ampliar los conocimientos adquiridos, ha sido la siguiente:

- International Standard ISO/IEC 14772-1:1997
- Teach Yourself VRML 2 in 21 Days – *Chris Marrin & Bruce Campbell*
- Web Workshop, 3D Graphics & VRML 2.0 – *Laura Lemay, Justin Couch & Kelly Murdock*
- Special Edition Using VRML - *Stephen Matsuba & Bernie Roehl*
- VRML Interactive Tutorial – Software, Interaction & Multimedia – Departamento de Informática - *Universidade do Minho – Portugal*
- Arquitectura del PC – Volumen I: Microprocesadores – *Manuel Ujaldón Martínez*
- CALCULO – Quinta Edición – Vol 1 y 2 – *Larson / Hostetler / Hedwards*
- ECMAScript 4 Netscape Proposal – *Waldemar Horwat*
- JavaScript 2.0 – *Waldemar Horwat*
- Proposal for a VRML Script Node Authoring Interface – VRMLScript Reference – *Chris Marrin, Jim Kent – Silicon Graphics, Inc.*
- Apuntes de la asignatura Traductores Compiladores e Intérpretes – *Dr. Sergio Gálvez Rojas*

MUNDOS VIRTUALES 3D CON VRML97

Daniel Héctor Stolfi Rosso

Sergio Gálvez Rojas



Desde las primeras esculturas prehistóricas en barro hasta el realismo conseguido por Fidias en la antigua Grecia o por Miguel Ángel con sus magníficas obras en mármol, el Hombre ha intentado desde muy antiguo representar en tres dimensiones el mundo que le rodea con mayor o menor precisión.

Pero estas representaciones estáticas han sufrido una enorme revolución con el advenimiento de los modernos ordenadores de hoy día y sus impresionantes posibilidades gráficas. En el presente volumen el lector podrá encontrar los métodos con los que realizar sus propias obras virtuales mediante el lenguaje VRML97, mundos en 3D en los que poderse sumergir a través de la pantalla del ordenador y recorrer por todos sus rincones como si de verdad existiesen. El nivel de detalle que puede alcanzarse dependerá únicamente del *constructor del mundo*, ese arquitecto en el que se convierte el programador cuando toma bajo su control este lenguaje para crear mundos virtuales en tres dimensiones.

Este texto proporciona los conceptos geométricos básicos 3D de VRML97 (cilindros, esferas y prismas) e introduce progresivamente otros más elaborados que permiten refinar una escena 3D tales como puntos de luz, texturas y ambientes. Se estudian asimismo los mecanismos de escena dinámicos propios de VRML97: sonidos, movimiento de objetos, colisiones y sensores que interaccionan con el usuario (apertura de puertas, viaje en ascensor o incluso permitir el hacer pintadas en una pared). Para finalizar, también se orienta al programador sobre cómo agrupar los objetos que se repiten en una escena tales como árboles, puertas y demás mobiliario, habitaciones en un bloque de apartamentos, etc. y todo ello, además, con el objetivo de construir mundos de un tamaño llevadero que facilite su labor a los motores de renderizado de VRML97.

Con estas herramientas el lector puede convertirse en un poderoso creador de mundos en los que la paciencia y la inventiva serán sus mejores aliados.

ISBN 978-84-693-0321-4

9 788469 303214

UNIVERSIDAD DE MÁLAGA