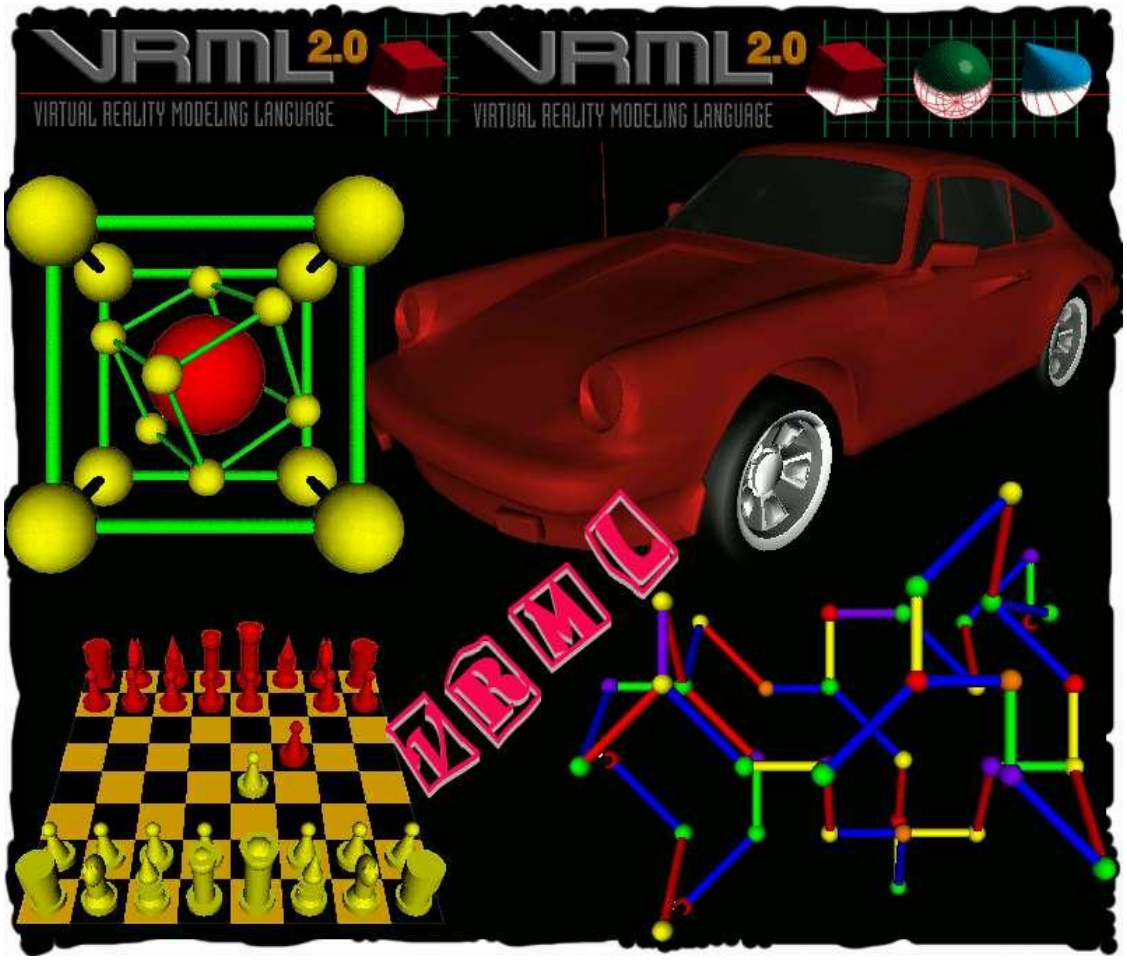
CAPITULO III

VRML(Lenguaje para Modelado de Realidad Virtual)



3.1. INTRODUCCION

VRML (Lenguaje de Modelado de Realidad Virtual) es un lenguaje de descripción de escenas con el que se pueden desarrollar escenarios interactivos en tres dimensiones 3D accesibles a través de Internet. Estos escenarios constituyen lo que se denomina realidad virtual interactiva, pues los usuarios pueden interactuar con los objetos de una forma similar a como lo hacen en la realidad "normal".

La realidad virtual esta revolucionando de manera que los usuarios se relacionan con sus ordenadores, de modo similar a lo ocurrido con el World Wide Web. Las posibilidades son innumerables: simulaciones educativas, nuevos métodos de organizar la información, nuevas formas de entretenimiento, etc.

En este capítulo se describe una reseña histórica de VRML 2.0, sus características, funcionamiento, browsers o plug – in y sintaxis de sus funciones para el diseño y navegación de escenarios virtuales.

VRML es un lenguaje de modelado de Realidad Virtual o descripción de escenas que describe simulaciones interactivas, escenarios virtuales accesibles a través de Internet combinando objetos realizados en el World Wide Web y VRML.

3.2. RESEÑA HISTORICA

[1]En 1.989, Rikk Carey y Paul Strauss de Silicon Graphics Inc, iniciaron un proyecto con el fin de diseñar y construir una infraestructura para aplicaciones interactivas con gráficos tridimensionales.

La primera fase del proyecto se concentraba en diseñar y construir la semántica y los mecanismos para la plataforma de trabajo. En 1.992 se liberó el Iris Inventor 3D Toolkit que fue el primer producto de dichos esfuerzos, este define gran parte de la semántica que hoy en día conforma a VRML.

La idea global surgió en la primera conferencia World Wide Web realizada en Ginebra en la primavera de 1994, durante una sesión de trabajo organizada por Tim Berners - Lee, creador de HTML y padre del WWW y Dave Raggett., con el objeto de discutir acerca de la posibilidad de incorporar una interfaz de realidad virtual para el WWW. Entre los asistentes se determinó la necesidad de disponer de un lenguaje simple capaz de describir una escena tridimensional, distribuible a través de HTTP e integrable en el mundo WWW mediante la incorporación de

[1] REF: http://www.vrml.org/VRML/FINAL

hiperenlaces que permitieran saltar a otras escenas 3D o a documentos HTML. Así, surgió el nombre de VRML, Virtual Reality Markup Language, por analogía con el de HTML (HyperText Markup Languaje). Posteriormente el nombre se cambio por el de Virtual Reality Modeling Language, mucho más apropiado. En la misma sesión se decidió comenzar el proceso de especificación del mismo y crear la lista de correo www-vrml con este objetivo. Al final de su primera semana de existencia, la lista ya contaba con más de mil suscriptores.

Poco tiempo después Mark Pesce, moderador de la lista, anunció el objetivo de disponer de una primera especificación del lenguaje para invierno de ese mismo año.

Lo primero que se hizo fue una revisión de todas las soluciones ya existentes encontrándose especialmente apropiado el formato de ficheros de Open Inventor, un Toolkit orientado a objetos Iris Inventor 3D Toolkit, para el desarrollo de aplicaciones gráficas interactivas propiedad de Silicon Graphics Inc, se sugirió a esta empresa que hiciera el formato libre y SGI no sólo aceptó sino que, tomo un subconjunto del formato, lo modificó de acuerdo a lo discutido en la lista de correo y se le añadieron las extensiones para red. De esta forma, surgió la primera versión de VRML que daría lugar a VRML 1.0.

SGI declaró el formato de uso público y cedió un parser al dominio público para el rápido desarrollo de visores VRML.

3.3. ¿ QUE ES VRML?

[2]Técnicamente hablando, VRML (Lenguaje para Modelado de Realidad Virtual), provee un conjunto básico de primitivas para el modelaje geométrico tridimensional y tiene la capacidad de dar comportamiento a los objetos y asignar diferentes animaciones que pueden ser activadas por eventos generados por diferentes usuarios.

El lenguaje VRML es un entorno de programación que permite crear escenarios virtuales constituidos por elementos generados en 3D, es decir, es un lenguaje de descripción de escenas, que describen simulaciones interactivas, escenarios virtuales accesibles a través del Internet y vinculados con elementos Web. Todos los aspectos de presentación de escenarios virtuales, interacción y conexión a la red pueden ser especificados usando VRML.

A pesar de que en estos momentos el lenguaje más extendido en la red es el HTML, el VRML presenta una importante ventaja respecto a él, puesto que permite a los usuarios visualizar los escenarios tal como sus mentes lo harían, es decir, en 3D. Es por esta razón que las páginas Web programadas en realidad virtual muestran mayor atractivo y una importante ventaja con respecto a las representaciones planas, a las que tiene acostumbrado Internet, disponiendo de

[2] REF: http://www.vrml.org/VRML/FINAL; http://www.activamente.com

un interesante futuro. Una de sus principales características es ésta, su capacidad de sorprender. Por medio de los escenarios virtuales puede visitar palacios italianos, aplicaciones de carácter científico, galerías comerciales, etc.

Al igual que HTML, VRML utiliza browsers para ver los escenarios virtuales. Un browser de VRML carga un escenario virtual descrito en VRML y luego lo muestra, dibujando una gráfica en la pantalla de su computadora, en 3D, permitiéndole viajar a través del escenario virtual. Se pueden seleccionar links en el espacio virtual que lo puedan llevar a otros escenarios o a cualquier URL.

Los requerimientos técnicos para aprovechar la tecnología VRML son cada vez más sencillos, gracias no sólo al avance de sus desarrolladores, sino también a la evolución de sus usuarios y del equipo de cómputo que cada vez más personas utilizan en el mundo moderno.

[3]Aunque el VRML es un lenguaje de computadora, no es un lenguaje de programación: los archivos VRML no se compilan, son archivos simples de texto que pueden ser leídos por un interpretador de VRML. Estos programas se denominan visualizadores (browser) de VRML. Ya que él código VRML se interpreta, los resultados visibles cambian de visualizador a visualizador, de acuerdo a la forma en que se encuentren implementados.

3.4. ¿QUE SE NECESITA PARA NAVEGAR EN VRML?

[4]Gracias a que VRML fue desarrollado para que millones de personas puedan interactuar, casi cualquier usuario puede accesar sitios producidos en VRML. Contrario a lo que se piensa, los espacios de realidad virtual se descargan muy rápidamente del Web, reduciendo el tiempo de espera enormemente comparado con su contra parte el HTML (Lenguaje utilizado para el desarrollo de páginas convencionales).

Los Navegadores actuales ya tienen instaladas diferentes versiones de Accesorios para VRML, por lo que sí tiene un navegador actualizado podrá ver escenarios VRML sin la necesidad de descargar ningún complemento especial.

Los requerimientos para visualizar escenarios virtuales son:

- Para los navegadores antiguos existen varios tipos de accesorios para la navegación en VRML e incluso hay algunos navegadores diseñados únicamente para navegar escenarios virtuales. Los accesorios son instalables en el navegador (visores o browser)

[3] REF: Mark Pesce, Pag 331.
[4] REF: http://www.vrml.org/VRML/FINAL

existente y por lo general son gratuitos. Recomendado: Cosmo Player de Silicon Graphics.

Algunos de los navegadores de VRML más importantes son:

- Cosmo Player
- Live3D
- Liquid Reality
- Community Place

Puede respirar tranquilo, no necesita comprar una tarjeta, guantes especiales ni lentes de Realidad Virtual para poder entrar en estos entornos.

Los visores VRML originales eran programas independientes del browser WWW tradicional, que únicamente era capaz de presentar documentos HTML, pero hoy en día, la mayoría suelen encontrarse integrados en los browsers WWW o se pueden incorporar a éstos en forma de Plug - Ins. Existe multitud de browsers para la versión 1.0 de VRML, sin embargo, es más difícil encontrar clientes para la versión 2.0.

- Muchos creen que la realidad virtual no es para ellos, porque no poseen una conexión muy buena a Internet. Sin embargo VRML fue diseñado precisamente para ser usado a través de Internet, usando el menor ancho de banda (o la menor conexión) posible y aprovechando al máximo los recursos del equipo cliente (del usuario). En realidad VRML puede desplegar más datos en menos tiempo, utilizando conexiones limitadas. Por eso una conexión telefónica con un módem de 14.4 Kbps es más que suficiente para visitar escenarios VRML.

- Las computadoras comerciales que se encuentran hoy en día son suficientes para navegar escenarios hechos en VRML. Lógicamente una computadora rápida permite una visualización más real y con mayor detalle. También influye el diseño del escenario virtual tanto en el tiempo de carga como en la visualización en tiempo real. El número de polígonos utilizados en el modelaje de los objetos virtuales, y la cantidad de gráficas o sonidos que se empleen en dichos escenarios son directamente proporcionales al tiempo de cálculo y de carga respectivamente.

  Los requerimientos mínimos están cercanos a un procesador Pentium de 75MHz con 32 MB en RAM o su equivalente en otras plataformas.

3.5. CARACTERISTICAS DEL LENGUAJE VRML

VRML define propiedades y relaciones entre objetos. Estos objetos pueden, teóricamente, tomar cualquier forma concreta, una geometría 3D, una imagen, un sonido, texto, etc. Para VRML todos los objetos se denominan genéricamente nodos.

Los nodos se organizan genéricamente en grafos de escena. Estos grafos de escena no definen únicamente conjuntos de nodos sino que aportan a estos conjuntos las nociones de orden y ámbito. De esta forma, cada nodo puede afectar los nodos que aparecen después en su mismo ámbito, en su grafo de escena. Por ejemplo un cono podría encontrarse rotando y dotado de una determinada textura si previamente se hubieran definido los nodos correspondientes a la rotación y textura en su mismo ámbito. Por ello, para aislar los nodos en diferentes ámbitos se definen también nodos separadores.

Un nodo VRML es un objeto que contiene los siguientes atributos:

- **Tipo de objeto:** Pueda ser un cubo, una esfera, una textura, una transformación, etc.
- **Campos:** Identifican las características de cada nodo, por ejemplo, el radio de una esfera o el mapa de texturas. Un nodo puede contener 0 ó más campos y éstos pueden ser escalares o vectoriales.
- **Nombre identificativo:** Cada nodo puede, opcionalmente, disponer de un único nombre. Este nombre debe ser único por nodo, pero dos o más nodos podrían compartir su nombre.
- **Nodos hijos:** La jerarquía de nodos se implementa permitiendo que los nodos puedan contener otros nodos. Los nodos pueden tener 0 ó más hijos, recibiendo en este último caso el nombre de nodos de grupo.
- VRML es capaz de representar objetos estáticos y puede tener hipervínculos a otros medios, tales como páginas Web
- Ofrece una experiencia más interactiva que HTML, ya que tiene una forma detallada de presentar y organizar la información, experimentando un espacio o entorno en tres dimensiones.
- Las herramientas de VRML permiten la creación de sofisticados escenarios virtuales sin profundizar en la complejidad del lenguaje. Sin embargo, VRML necesita computadoras veloces con modems rápidos para explorar escenarios virtuales de gran tamaño.

3.6. OPCIONES Y BROWSERS PARA REALIZAR NAVEGACIONES VIRTUALES EN ESCENARIOS VRML.

Existen dos opciones para visualizar escenarios virtuales:

- Utilizando un navegador especial, diferente a los tradicionales como Netscape o Internet Explorer.
- Mediante programas adicionales que se instalan dentro de los navegadores tradicionales.

Los programas adicionales, mejor conocido como Plug - In, son la alternativa más utilizada por ser más fáciles y cómodos de instalar.

3.6.1. Navegadores o Visualizadores

- Cosmo Player sus creadores son Silicon Graphics. Indudablemente la primera alternativa. Soporta las especificaciones VRML 2.0. Disponible para Windows 95/98 y NT.
- Cyber Passage 2.0 Beta 2 es de la Sony y soporta las especificaciones VRML 2.0. Disponible para Windows 95 y NT.
- WorldView 1.0 Beta 3 es de la empresa InterVista. Disponible para Windows 95 y NT.
- VR Scout Viewer es el visualizador particular de la misma empresa que realiza el plug - in. Disponible para Windows 3.1, Windows 95 y NT.
- Whurlwind para Mac PowerPC, requiere QuickDraw 3D.
- Community Place Sony's Virtual Society, para Win95 y NT.
- Liquid Reality (browser y toolkit), para Win95 y NT.
- GL View OGL/D3D(para OpenGL o Direct3D), para Win95 y NT.

Una vez instalado cualquiera de estos plug - in o navegadores, es obviamente necesario que lo pruebe con una página diseñada en VRML.

Una vez dentro de un escenario virtual nadie le dice que hacer, tiene la libertad de acercarse o alejarse de los objetos, moverlos, verlos desde cualquier perspectiva posible e incluso ingresarte en los objetos.

3.6.2. Herramientas para VRML 2.0.

Análogamente a lo que ocurre con los visores, existe una gran cantidad de herramientas para la edición de VRML 1.0 y no es difícil encontrar conversores para la mayoría de los formatos 3D existentes. Sin embargo, el número se reduce drásticamente cuando se buscan herramientas para VRML 2.0, hasta tal punto que únicamente se puede citar tres:

- ParaGraph International. Virtual Home Space Builder VRML 2.0 authoring tool. (PC & MAC).
- Integrated Data Systems. VRealm Builder VRML 2.0 authoring tool. (WIN 95 & NT, PC &DEC).
- Silicon Graphics, Inc. Cosmo Worlds VRML 2.0 authoring tool. (IRIX).

3.7. ESTANDARES

Existen dos estándares oficiales al momento: VRML 1.0. y VRML 2.0.

[5]La versión 2.0. es el resultado de la votación del comité de expertos celebrada en Agosto de 1996, se adoptó la propuesta presentada por la compañía Silicon Graphics bajo el nombre de "Moving Worlds" (mundos en movimiento), pasando a ser el actual estándar oficial con el nombre de VRML 2.0, y venciendo a otras propuestas presentadas por compañías del calibre de Microsoft, Apple y Sun Microsystems.

Esta nueva versión (VRML 2.0), es más sofisticada que la anterior: los objetos pueden tener comportamientos propios, especificados incluso con scripts en otros lenguajes distintos (JavaScript, Java, Visual Basic, etc.), lo que le confiere una enorme flexibilidad.

Otro aspecto importante en esta versión es que se ha potenciado la interactividad del usuario con el entorno: se pueden definir sensores (de posición, colisión, contacto, etc.) que informan de lo que está haciendo el usuario para que los objetos puedan actuar en secuencia.

Además, para añadir realismo a los escenarios tridimensionales, se pueden crear fondos gráficos, efectos de niebla, sonidos tridimensionales, etc.

Este estándar (VRML 2.0), en su forma final, también recibe el nombre de VRML 97.

[5] REF: http://www.vrml.org/VRML/FINAL

3.8. BROWSER COSMO PLAYER 2.1.1.

[6]Antes de empezar con el estudio de la sintaxis del lenguaje VRML se mostrara un resumen de los controles de Cosmo Player 2.1.1, ya que este browser servirá para ir visualizando los resultados y efectos de los documentos VRML presentados a continuación

Para moverse en el escenario virtual o mover el objeto se debe pulsar el ratón y arrastrarlo, o usar las teclas de desplazamiento o flechas. El tipo de movimiento se hará según los mandos seleccionados.

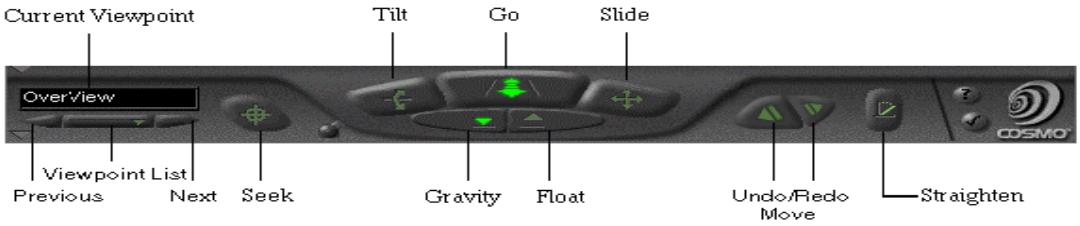


Figura 3.1. Controles del Cosmo Player para ver escenarios en 3D

Tilt (inclinar): para mirar alrededor del objeto, sin moverse.

Go (ir): arrastrar el ratón para moverse en el escenario (hacia adelante, atrás, a la derecha o a la izquierda).

Slide (deslizarse): para moverse en un mismo plano vertical.

Seek (buscar): pulse este botón primero, y luego al pulsar directamente en el objeto sirve para ver el objeto más de cerca.

Gravity (gravedad): permite moverse en un escenario por el suelo.

Float (volar): permite flotar por encima del suelo.

Undo/Redo Move (deshacer/repetir movimiento): para volver o repetir movimientos anteriormente efectuados.

Straighten (enderezar): vuelve a poner en posición vertical el escenario virtual.

Pulsando el botón **Change Controls** aparecen otros mandos o controles:

[6] REF: http://www.cosmosoftware.com

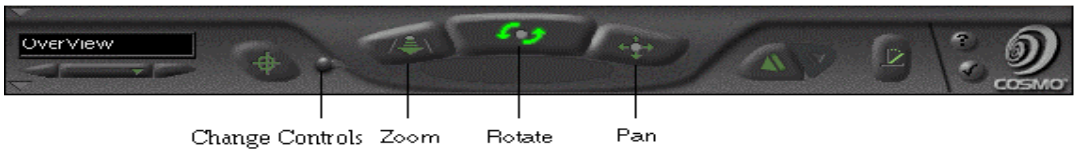


Figura 3.2. Controles del Cosmo Player para examinar el Escenario

Zoom: para acercar o alejar un objeto.

Rotate (girar): gira el objeto alrededor de un punto.

Pan: mueve el objeto en un plano.

Viewpoint List: se encuentra al lado izquierdo, al presionar este botón restaura el objeto o escenario virtual a su punto de vista original.

SINTAXIS DEL LENGUAJE VRML

3.9. ESTRUCTURA DE LOS DOCUMENTOS VRML

[7]VRML es un lenguaje de descripción de escenas en el que cada escena se compone de un número de objetos. Los objetos pueden ser formas sólidas situados y orientados de determinada forma u elementos intangibles que afectan a la escena como luces, sonido y distintos puntos de vista.

Para crear estos escenarios de realidad virtual se utilizan ficheros de texto, cuya extensión será siempre .wrl o .wrz, los cuales pueden ser desarrollados mediante cualquier editor o procesador de textos.

Además, existe la posibilidad de utilizar programas de diseño gráfico, los cuales generan automáticamente ficheros en formato VRML.

En líneas generales, un documento VRML contiene los siguientes elementos:

- Línea de cabecera.
- Comentarios al código.
- Nodos.

[7] REF: http://www.publimas.com/tutoriales/vrml/documentos/tema1.htm

3.9.1. Líneas de Cabecera

Todo documento VRML debe comenzar necesariamente con la siguiente línea:

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

Con esta línea inicial se hace la declaración de que el estándar empleado es el VRML 2.0. El identificativo utf8 sirve para permitir el uso de caracteres internacionales.

No se debe dejar ningún espacio en blanco antes del comienzo de la línea (antes del símbolo **#**), pues en caso contrario el visualizador no la reconocería, y daría error al interpretar o ejecutar el documento VRML.

3.9.2. Comentarios al código

En todos los lenguajes se utilizan comentarios, no visibles en la pantalla, que son muy útiles como recordatorio de lo realizado, facilitando los cambios futuros.

En este lenguaje los comentarios se escriben en una sola línea que comienza con el símbolo **#**. Ejemplo:

```
# Este es un comentario al código
```

3.9.3. Nodos

Los nodos son bloques de información que definen las características de un objeto concreto (su forma, su apariencia, etc.), o la relación entre distintos objetos. Sirven también para crear sonidos, el panorama de fondo, enlaces a otros escenarios o a páginas Web, etc.

Es decir, son las unidades básicas que forman el escenario virtual, y pueden ser afinadas hasta el detalle deseado.

La sintaxis (reglas para la correcta escritura) de los nodos, desde un punto de vista general es:

```
Nombre {
    campo1 x y z
    campo2 x
    ...
}
```

- El nodo es un bloque de información que tiene un Nombre (Box, Cone, etc.). Obsérvese que comienzan siempre por una letra mayúscula.
- Englobados entre los símbolos { y } el nodo tiene uno o varios campos (size para el nodo Box, height y bottomRadius para el nodo Cylinder, etc.). Los campos son los atributos variables del nodo. Obsérvese que comienzan con una letra minúscula.
- A continuación está el valor que le asigna a ese campo, que es un número (o conjunto de números) Estos números se escriben con punto flotante (es decir, con decimales).

  En ocasiones, en lugar de colocar como valor un número (o conjunto de números) se coloca todo un nodo. Esta noción, que puede parecer confusa de momento, se verá con detalle más adelante.

3.10. NODOS PRIMITIVOS

[8]Los nodos son los bloques de información básicos con los que se construye un escenario virtual. Ahora se verá la sintaxis de los nodos, partiendo de los más sencillos, los nodos de geometría primitiva, o nodos primitivos, que son:

- Box caja
- Cone cono
- Cylinder cilindro
- Sphere esfera

Con estas figuras geométricas básicas, agrupándolas y modificándolas convenientemente se pueden construir formas de gran detalle.

Para formas realmente complejas, hay otros procedimientos para crear objetos, partiendo de puntos, líneas o caras

3.10.1. Box

La estructura más general de este nodo es:

```
Box {
    size anchura altura profundidad
}
```

[8] REF: http://www.publimas.com/tutoriales/vrml/documentos

- Se escribe el nombre del nodo, en este caso Box, seguido del símbolo de apertura {.
- Luego se escribe el campo, que es un atributo del nodo que se puede cambiar a voluntad, poniendo los valores que se desee. En este caso servirá para establecer las dimensiones de la caja, como se verá a continuación.
- Posteriormente se escribe el símbolo de cierre }.

Con respecto a los nombres, y en general cualquier otro comando usado en este lenguaje, se debe tener muy en cuenta que son sensibles a las mayúsculas y minúsculas, es decir, se debe escribir Box y no box, o fontStyle y no fontstyle, etc.

Sustituyendo ahora los campos que define este nodo, que es size (tamaño) con sus valores correspondientes (las dimensiones), queda el nodo completo de esta forma:

```
Box { size 2 0.5 3 }
```

Los números representan respectivamente la anchura, la altura y el fondo. Son dimensiones abstractas, pero es conveniente, aunque no obligatorio, suponer que son metros. El nodo, por tanto, define una caja de 2 m de ancho, 0.5 m de alto y 3 m de fondo.

Se ha escrito el nodo en una sola línea, debido a su sencillez y a que sólo tiene un campo, pero se podría haber escrito de esta otra forma:

```
Box {
     size 2 0.5 3
}
```

3.10.2. Cone

Sintaxis:

```
Cone {
    height altura
    bottomRadius radio_de_la_base
    bottom valor_lógico
    side valor_lógico
}
```

Mediante los campos bottom y side se indican si se desea dibujar la base y la superficie lateral respectivamente. Por defecto estos campos toman el valor **TRUE**, lo cual indica que se dibuja el cono completo.

Un ejemplo de este nodo, que define la geometría de un cono:

```
Cone {
    height 3
    bottomRadius 0.75
}
```

Como se puede observar, tiene dos campos: height (altura), a la que se le ha puesto el valor de 3, y bottomRadius (radio de la base) el de 0.75. Con estos dos valores queda perfectamente definido el cono.

3.10.3. Cylinder

Sintaxis:

```
Cylinder {
    height altura
    radius radio
    bottom valor_lógico
    side valor_lógico
    top valor_lógico
}
```

Mediante los campos bottom, side y top se indica si se desea dibujar la base inferior, la superficie lateral y la base superior del cilindro. Por defecto estos campos toman el valor TRUE, lo cual indica que se dibuja el cilindro completo.

Ejemplo del nodo que define un cilindro:

```
Cylinder {
    height 3
    radius 1.5
}
```

Tiene sus dos campos: height (altura) y radius (radio)

3.10.4. Sphere

Sintaxis:

```
Sphere {
    radius radio
}
```

Ejemplo del nodo que define una esfera:

```
Sphere {
    radius 2
}
```

Tiene un solo campo, radius (radio) este es suficiente para definir con él una esfera.

Estos cuatro nodos de geometría primitiva, no se pueden utilizar directamente ellos solos, sino que son parte de otro nodo más general, el nodo Shape (forma).

El motivo de esto es que, definen únicamente la geometría de estos cuerpos, pero no dan ninguna indicación de cuál deberá ser su apariencia, es decir, su color, textura, iluminación, etc.

El nodo Shape se encarga de ambas cosas, ya que tiene dos campos: la apariencia y la geometría, en donde utiliza precisamente estos nodos que se acaba de ver, es decir, estos nodos irán incrustados (o como parte) en el nodo Shape.

3.11. CONSTRUCCION DE FORMAS DE TEXTO

En los escenarios virtuales es a menudo necesario utilizar textos para guiar al visitante, para ello existe un nodo específico, el nodo Text. Una de las principales características de los textos es que son planos, es decir, no tienen profundidad.

3.11.1. Nodo Text

Como en cualquier procesador de textos, se permitirá indicar el tipo de fuente, su estilo, su tamaño, el espaciado entre caracteres, justificación de los párrafos, etc. Su sintaxis es:

```
Text {
    string ["linea_texto 1",
    "linea_texto 2",
                ...
    "linea_texto N",]
    fontStyle FontStyle { ... }
}
```

Como se puede apreciar el nodo Text posee dos campos:

string	Sirve para introducir el texto que se desea visualizar.
fontStyle	Este segundo campo es opcional, de forma que si se omite, el texto tendrá el estilo de la fuente por defecto.

Siempre que aparezca este campo tomará como valor el nodo llamado FontStyle.

3.11.2. Nodo FontStyle

Este nodo permite seleccionar el tipo letra, tamaño, estilo, alineación y espacio de separación entre letras: Su sintaxis:

```
FontStyle {
    family "Nombre_Fuente",
    style "Estilo_Fuente",
    size Tamaño_Fuente
    spacing espaciado_entre_caracteres
    justify "justificación_del_texto"
}
```

Los posibles valores de los campos del ***nodo FontStyle*** son los que se explican a continuación:

family	Determina la fuente que se va a utilizar para el texto. Se puede escoger entre "SERIF", "SANS" o "TYPEWRITER". Los nombres de las fuentes deben estar en mayúsculas.
style	Contiene los siguientes parámetros "BOLD" (negrita), "ITALIC" (cursiva), "BOTH" (negrita y cursiva) o "NONE" (tipo de letra normal).
size	Determina el tamaño de la fuente, pero en unidades VRML.
spacing	Determina la separación entre líneas, también en unidades VRML.
justify	Determina la justificación del texto. Puede ser "BEGIN" (Alinear a la izquierda), "MIDDLE" (centrar el texto) o "END" (Alinear a la derecha).

Todos estos campos son opcionales. Ejemplo:

```
Text {
    string ["Esta es la primera fila de texto",
    "esta es la segunda fila",
    "etc."]
    fontStyle FontStyle {
        family "SERIF",
        style "BOLD",
        size 1.0
        spacing 1.0
        justify "BEGIN"
    }
}
```

De igual forma que con los nodos primitivos, el nodo Text lo único que consigue es definir la estructura del texto, sin embargo no se puede visualizar. Para conseguir esto, se integra en el nodo Shape.

3.12. NODO Shape

Para crear un objeto visible se debe utilizar el nodo Shape, tiene dos campos: appearance y geometry. Con el primero se controla la apariencia del objeto (color, textura, iluminación, etc.) y con el segundo su geometría. La estructura general de Shape es:

```
Shape {
    appearance ...
    geometry ...
}
```

Los puntos suspensivos representan los valores de cada campo.

El primer campo (appearance) es opcional, y se puede prescindir de él por el momento, es decir, el estudio se lo va a ser de la siguiente forma:

```
Shape {
    geometry ...
}
```

El valor del campo geometry (los puntos suspensivos) es precisamente uno de los cuatro nodos primitivos vistos anteriormente (Box, Cone, Cylinder y Sphere o Text).

Se utilizará el nodo Box para ponerlo como valor del campo geometry*:*

```
Shape {
    geometry Box {
          size 2 0.5 3
    }
}
```

Con esto se define un objeto visible (sin ningún atributo de apariencia), que tiene la geometría de una caja de medidas 2 x 0.5 x 3

Ahora ya está en condiciones de crear el primer documento VRML con el que se podrá ver el objeto con el visualizador, aunque todavía en unas condiciones bastante imperfectas, ya que no tiene definido ningún atributo de apariencia.

Para crear el documento VRML sólo falta ponerle la línea de cabecera, y algún comentario si desea:

```
#VRML V2.0 utf8
#Progra1.wrl
#Primer documento VRML
#Caja sin apariencia definida
Shape {
    geometry Box {
        size 2 0.5 3
    }
}
```

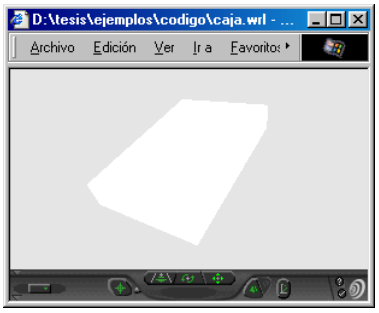


Figura 3.3 Forma primitiva Box.

Al lado izquierdo se encuentra el código VRML, el mismo que, se puede grabar con el nombre que se desee con la extensión .wrl, su nombre será Progra1.wrl, para ver el resultado se utiliza el visualizador o Browser Cosmo Player 2.1.1, que debe estar instalado en su computadora.

Como se puede observar en la imagen, las caras del objeto son de color blanco y no se distinguen unas de otras, pudiéndose apreciar solamente el contorno del objeto.

Ahora se va a suministrar al objeto cualidades de apariencia que existen por defecto, para esto se utiliza el campo appearance, del que se había prescindido en el ejemplo anterior:

```
appearance ...
```

Se colocara un valor en los puntos suspensivos al campo appearance. No va a ser un número (o un conjunto de números), sino un nodo llamado Appearance.

Obsérvese la A mayúscula de Appearance, que indica que se trata de un nodo, mientras que en el caso de appearance la a minúscula indica que es un campo (del nodo Shape).

Observe, la estructura de este nuevo nodo Appearance:

```
Appearance {
    material Material {   }
}
```

Este nodo Appearance tiene, a su vez, un campo llamado material, cuyo valor es otro nodo, llamado Material, en el que se ha dejado en blanco su contenido, definiendo las características por defecto de este nodo. Después se aprenderá a manipular el nodo Material, para definir sus características a gusto (color, luminosidad, transparencia, etc.).

Centrándose en el caso concreto de la caja, se ensamblará las distintas piezas. Primero, se coloca el nodo Appearance como su respectivo campo appearance:

```
appearance Appearance {
    material Material {   }
}
```

Ahora puede colocar este campo dentro del nodo Shape, en el documento VRML, con algunos comentarios que faciliten su comprensión como muestra el siguiente ejemplo Progra2.wrl, con su código:

```
#VRML V2.0 utf8
#Progra2.wrl
#Caja con la apariencia por defecto
Shape {
    #Campo appearance:
    appearance Appearance {
        material Material {   }
    }
    #Campo geometry:
    geometry Box {
        size 2 0.5 3
    }
}
```

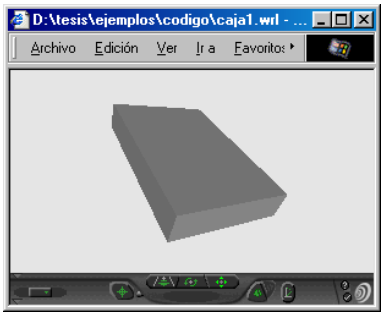


Figura 3.4. Utilización del Nodo Appearance

Como se puede apreciar en la Figura (y aún más manipulándola con el visualizador), la diferencia es notable. Ahora se distinguen perfectamente las caras de la caja, debido a que tienen una iluminación y textura adecuadas.

El mismo procedimiento se realiza con el nodo primitivo Cone:

```
#VRML V2.0 utf8
#Progra3.wrl
#Cono con la apariencia por defecto
Shape {
    #Campo appearance:
    appearance Appearance {
        material Material {   }
    }
    #Campo geometry:
    geometry Cone {
        height 3
        bottomRadius 0.75
    }
}
```

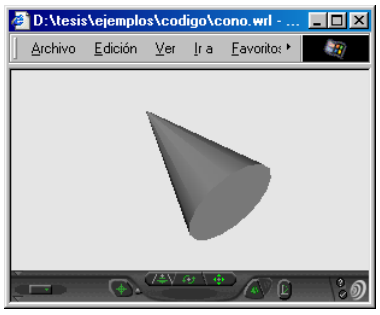


Figura 3.5. Nodo primitivo Cone

Como se puede observar, el campo appearance es idéntico al caso anterior, sólo cambia el valor del campo geometry, que es el nodo Cone.

Para completar los cuatro casos de geometría primitiva vistos anteriormente, se ha elaborado los ejemplos Progra4.wrl y Progra5.wrl y Progra6.wrl:

```
#VRML V2.0 utf8
#Progra4.wrl
#Cilindro con la apariencia por defecto
Shape {
    #Campo appearance:
    appearance Appearance {
        material Material {   }
    }
    #Campo geometry:
    geometry Cylinder {
        height 3
        radius 1.5
    }
}
```

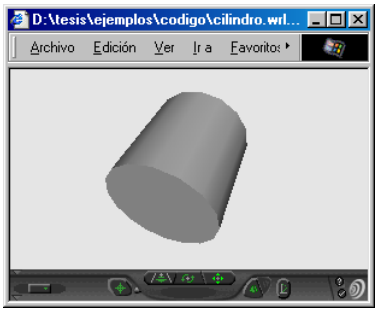


Figura 3.6. Nodo Primitivo Cylinder

```
#VRML V2.0 utf8
#Progra5.wrl
#Esfera con la apariencia por defecto
Shape {
    #Campo appearance:
    appearance Appearance {
        material Material {   }
    }
    #Campo geometry:
    geometry Sphere {
        radius 2
    }
}
```

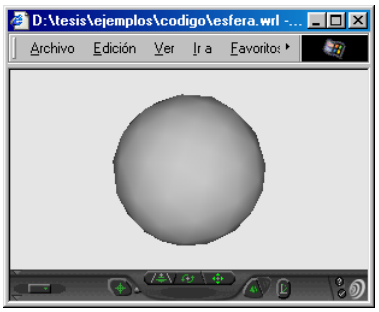


Figura 3.7. Nodo Primitivo Sphere

```
#VRML V2.0 utf8
#Progra6.wrl
#Ejemplo utilizando el nodo Text
Shape{
    appearance Appearance {
        material Material {}
    }
    geometry Text {
        string ["Esta es la primera fila de texto",
        "esta es la segunda fila",
        "etc."]
        fontStyle FontStyle {
            family "SERIF",
            style "BOLD",
            size 2.0
            spacing 1.0
            justify " MIDDLE "
        }
    }
}
```

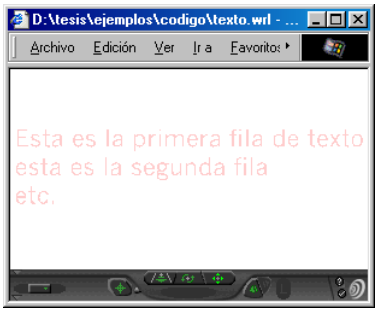


Figura 3.8. Nodo Text

3.13. AGRUPACION DE NODOS

Hasta ahora sé ha visto los objetos aisladamente. Cómo se pude agrupar un conjunto de ellos para conseguir formas más complejas, los nodos para agrupar objetos son:

- Nodo Group.
- Nodo Transform.
- Nodo Switch.
- Nodo Billboard.
- Nodo Anchor.
- Nodo Collision.

3.13.1. Nodo Group

El nodo ***Group*** permite tratar a un conjunto de nodos como una entidad única, pero sin efectuar o realizar ninguna transformación en ellos.

Su sintaxis o estructura general es la siguiente:

```
Group {
    children [ ... ]
}
```

Tiene un campo llamado children (niños o hijos), cuyo valor (los puntos suspensivos) va entre corchetes [y], los puntos representan la lista de los objetos que se quieren agrupar, representados por sus nodos Shape respectivos:

```
Group {
    children [
        Shape { ... },
        Shape { ... },
        ...
    ]
}
```

El ejemplo Progra7.wrl agrupa una caja con un cono:

```
#VRML V2.0 utf8
#Progra7.wrl
#Ejemplo de agrupación de una caja y un cono
Group {
    children [
        #Aquí empieza la caja
        Shape {
            appearance Appearance {
```

```
            material Material {   }
        }
        geometry Box {
            size 2 1 3
        }
    },
    #Aquí empieza el cono
    Shape {
        appearance Appearance {
            material Material {   }
        }
        geometry Cone {
            height 4
            bottomRadius 1
        }
    }
]
}
```

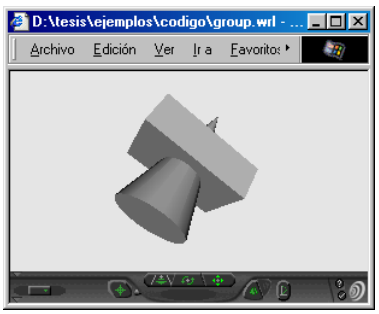


Figura 3.9. Ejemplo del Nodo Group

Vea el resultado en la imagen de la derecha. Se puede observar que la caja y el cono están superpuestos. Esto es debido a que los objetos son creados todos en el mismo punto: en el centro del escenario de realidad virtual.

Para colocar un objeto, o grupo de objetos, en otros puntos que no sean el centro, se deberá utilizar otro tipo de nodo, que se ve más adelante.

3.13.2. Consejos prácticos para escribir código VRML

Como sé ha visto la sintaxis de cada nodo y la manera de cómo se incrustan unos nodos dentro de otros, etc., para poder construir los correspondientes documentos VRML.

Si observa el último ejemplo, parece como si el código se complicara más y más a pesar de tratarse de un ejemplo relativamente sencillo.

No hay tal complicación, pero sí es muy importante seguir unas cuantas reglas prácticas, pues es absolutamente imprescindible que las cosas estén colocadas en el orden debido, y que no sobre ni falte ninguno de los símbolos de apertura y de cierre { y }, [y].

- Utilice una línea distinta para cada nodo, para cada campo y para cada valor de cada campo.

  No es que sea obligatorio, pues de hecho se podría escribir todo el código en una sola línea, aunque esto no sería muy práctico e induciría a cometer errores.

  Si el código del ejemplo anterior, se escribe en una sola línea se tendría que omitir las líneas de comentarios, pues éstas requieren forzosamente estar en una línea aparte

- Indentar cada línea, según su jerarquía.

Para indentar, es muy útil utilizar la tecla tabuladora, que desplaza el cursor un número fijo de espacios.

- Colocar cada símbolo de cierre en el nivel de indentación que le corresponda**.**

  Es práctico que los símbolos de cierre estén situados exactamente debajo del comienzo de la línea que le corresponde y al mismo nivel del nodo que cierren. De esta manera están perfectamente identificados, y sirve como control para no olvidarse de ninguno de ellos.

- Poner las líneas de comentario necesarias al mismo nivel que lo comentado.

3.13.3. Comandos DEF y USE

Observe en el ejemplo anterior, se ha repetido dos veces el nodo Appearance, para dotar de la misma apariencia por defecto a los dos objetos (la caja y el cilindro).

No es tan grave en este caso, pero imagínese que se quiere dotar de esa misma apariencia, por ejemplo a cinco objetos distintos. Entonces, habría que repetir cinco veces todo el nodo Appearance en cada uno de los cinco objetos, lo cual complicaría innecesariamente el código.

Existe una solución para estas repeticiones, ya que se puede definir un nodo que se piensa repetir en el código, poniéndole un nombre arbitrario.

Suponga, por ejemplo, que se van a utilizar repetidamente en un escenario unos cilindros exactamente iguales (que podrían ser las columnas de una casa), y que dichos cilindros tienen el siguiente código:

```
Shape {
    appearance Appearance {
        material Material { }
    }
    geometry Cylinder {
        height 2
        radius 0.5
    }
}
```

Este es el código de un cilindro con la apariencia por defecto, de 2 unidades de alto y una base de radio 0.5.

Se puede definir, para el ámbito de un documento VRML, que este tipo de cilindro tenga un nombre arbitrario, por ejemplo ColumnaRepetida o cualquier otro nombre, lo importante es que comience por una letra mayúscula de la siguiente manera:

```
DEF ColumnaRepetida Shape {
    appearance Appearance {
        material Material { }
    }
    geometry Cylinder {
        height 2
        radius 0.5
    }
}
```

Entonces, cada vez que se quiera usar este nodo en otra parte del documento, basta con poner lo siguiente:

```
USE ColumnaRepetida
```

En el ejemplo de la caja y el cono agrupados, lo que está repetido es el nodo Appearance. Se definirán, en la primera ocasión se utiliza con el nombre por defecto y en la segunda vez se utilizará sólo el comando USE:

```
#VRML V2.0 utf8
#Ejemplo de agrupación de una caja y un cono,
#y utilización de los comandos DEF y USE.
Group {
    children [
        Shape {
            appearance DEF PorDefecto Appearance {
                material Material {   }
            }
            geometry Box {
                size 2 0.5 3
            }
        },
        Shape {
            appearance USE PorDefecto
            geometry Cone {
                height 3
                bottomRadius 0.75
            }
        }
    ]
}
```

En este caso concreto, la simplificación no ha sido muy grande, sólo un par de líneas menos de código, pero ha servido para ilustrar el concepto.

3.13.4. Ejemplo práctico con el nodo Group y comandos DEF y USE

Se realizará una pieza que esta compuesta por los siguientes objetos:

- Una caja de 10x10x10 (un cubo).

- Una esfera de radio 7.
- Un cilindro de altura 0.5 y radio de la base 12.5.
- Otro cilindro de altura 20 y radio de la base 4.
- Otro cilindro de altura 30 y radio de la base 3.
- Otro cilindro de altura 60 y radio de la base 1.

El código del Progra8.wrl es el siguiente:

```
#VRML V2.0 utf8
#Progra8.wrl
#Aplicación de Group y comandos DEF y USE
Group {
    children [
        Shape{
            appearance DEF Fondo Appearance {
                material Material {   }
            }
            geometry Box {
                size 10 10 10
            }
        },
        Shape{
            appearance USE Fondo
            geometry Sphere {
                radius 7
            }
        },
        Shape{
            appearance USE Fondo
            geometry Cylinder {
                height 0.5
                radius 12.5
            }
        },
        Shape{
            appearance USE Fondo
            geometry Cylinder {
                height 20
                radius 2
            }
        },
        Shape{
            appearance USE Fondo
            geometry Cylinder {
                height 30
                radius 3
            }
        },
        Shape{
            appearance USE Fondo
            geometry Cylinder {
                height 60
                radius 1
            }
        },
    ]
}
```

Figura 3.10. Ejemplo de los comandos USE y DEF

Al visualizar este ejemplo, puede ocurrir que inicialmente no se vea nada. La razón de esto es que algunos visualizadores (entre ellos el Cosmo Player) se sitúan en el centro mismo del escenario. En ese caso, es necesario retroceder, utilizando el mando Zoom.

Mas adelante se verá la manera de forzar al visualizador para que se sitúe en el punto que más convenga. El resultado de este código esta en la figura al lado derecho

3.13.5. Nodo Transform

Como se ha mencionado todos los objetos son creados en el centro del escenario de realidad virtual. Ahora se verá cómo ponerlos en otros puntos. Pero para ello, es necesario antes comprender la noción de los sistemas de coordenadas.

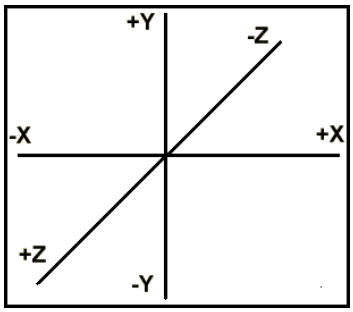


Figura 3.11. Sistema de Coordenadas

Un punto en el espacio está definido por su posición con respecto al centro de coordenadas. El lenguaje VRML adopta la convención de que sea X la distancia que desplaza al punto, a la derecha o a la izquierda del centro, Y la distancia por encima o por debajo, y Z la distancia hacia delante o hacia atrás.

Un escenario virtual tiene su sistema de coordenadas situado en el centro.

Con el nodo ***Group,*** visto anteriormente se consigue agrupar un conjunto de objetos, que son creados haciendo coincidir su centro con el centro del sistema de coordenadas

Con el nodo ***Transform***, que se verá a continuación, se determina un nuevo sistema de coordenadas para un grupo de objetos.

Este nuevo sistema de coordenadas sufre transformaciones: puede ser trasladado a un punto determinado, puede ser girado un determinado ángulo, y puede tener una escala (tamaño relativo) distinta a la original.

El grupo de objetos especificados en el nodo sufrirá las mismas transformaciones, es decir, serán trasladados, girados y variados de escala.

La estructura general del nodo Transform es la siguiente:

```
Transform {
        translation eje_x eje_y eje_z
        rotation eje_x eje_y eje_z ángulo
        scale eje_x eje_y eje_z
        children  [ . . . ]
}
```

Como se puede ver, tiene tres campos con los que se determina su posible traslado, rotación o nueva escala, y un cuarto campo children, en donde se especifican cuáles son los objetos agrupados que sufrirán esas transformaciones.

No tienen porque estar los tres campos a la vez. Por tanto, para simplificar, se verá cada uno de ellos por separado.

3.13.5.1. Traslación

Utilizando sólo el campo translation, queda el nodo Transform de esta manera:

```
Transform {
    translation 20 0 0
    children [ . . . ]
}
```

Se ha puesto un valor al campo *translation*. El orden para el traslado de las coordenadas es X, Y, Z. Por tanto, en este caso, hay una traslación de 20 unidades hacia la derecha (sí hubiera sido hacia la izquierda se habría puesto -20.0)

Por supuesto que se podrían haber variado las tres coordenadas, no sólo la X como en este caso. Con respecto al campo children, se aplica lo explicado para el nodo Group.

3.13.5.2. Rotación

Utilizando sólo el campo rotation, queda el nodo Transform de esta manera:

```
Transform {
      rotation 1 0 0 1.57
      children [ ... ]
}
```

Los parámetros del campo rotation se describen de la siguiente forma: rotación de 90° alrededor del eje X.

Las tres primeras cifras sólo pueden tener el valor 0 ó 1, y representan la rotación alrededor de cada eje, en este orden: X, Y, Z, es decir:

- Rotación alrededor del eje X: 1 0 0.
- Rotación alrededor del eje Y: 0 1 0.
- Rotación alrededor del eje Z: 0 0 1.

La cuarta cifra representa el ángulo girado, expresado en radianes.

Para calcular la correspondencia entre grados y radianes, hay que tener en cuenta que 180º equivalen al número pi (π) en radianes, es decir a 3.14 radianes. Por tanto, 90º sería la mitad de 3.14, es decir 1.57 radianes.

El sentido del giro es el de un sacacorchos avanzando en la dirección positiva de los ejes.

En el siguiente código se puede ver al grupo de objetos, trasladados 20 unidades hacia la derecha y girados 90º alrededor del eje X:

```
Transform {
    translation 20 0 0
    rotation 1 0 0 1.57
    children [ ... ]
}
```

3.13.5.3. Variación de la escala

Ahora utilizando en el nodo Transform el campo scale, queda de la siguiente forma:

```
Transform {
    scale 1 2 1
    children [ ... ]
}
```

Los valores del campo scale representan las variaciones de las dimensiones con respecto a los ejes X, Y , Z.

Por tanto, en el ejemplo que se ilustra, se reducen las dimensiones en la dirección de las X a la mitad (1), se duplican en la dirección del eje Y (2), y se reducen a la mitad en la dirección del eje Z (1).

Si se hubiera variado la misma cantidad en los tres valores (por ejemplo, 0.5) se habría mantenido las proporciones de la pieza.

Combinando los tres campos simultáneamente y aplicando al grupo de objetos del ejemplo anterior (traslación, giro y cambio de escala) el código del ejemplo Progra9.wrl es:

```
#VRML V2.0 utf8
#Progra8.wrl
#Aplicación de Transform
Transform {
    translation 20 0 0
    rotation 1 0 0 1.57
    scale 1 2 1
    children [
        Shape{
            appearance DEF Fondo Appearance {
                material Material {   }
            }
            geometry Box {
                size 10 10 10
            }
        },
        Shape{
            appearance USE Fondo
            geometry Sphere {
                radius 7
            }
        },
        Shape{
            appearance USE Fondo
            geometry Cylinder {
                height 0.5
                radius 12.5
            }
        },
        Shape{
            appearance USE Fondo
            geometry Cylinder {
                height 20
                radius 2
            }
        },
        Shape{
            appearance USE Fondo
            geometry Cylinder {
                height 30
                radius 3
            }
        },
        Shape{
            appearance USE Fondo
            geometry Cylinder {
                height 60
                radius 1
            }
        },
    ]
}
```

Figura 3.12. Ejemplo del Nodo Transform

Como se puede ver en la figura, el grupo de objetos después de sufrir los tres cambios y utilizar el control Zoom del visor.

En este ejemplo se ha utilizado un grupo de seis objetos definidos en el campo children. Pero dentro del campo children puede haber un solo objeto, no tienen por qué ser siempre varios objetos.

3.13.5.4. Ejemplo práctico con el nodo Transform

Se desarrollará un escenario virtual compuesto por planetas, en el cual se creará los siguientes objetos:

- Un planeta VRML, que es una esfera de 12 de diámetro, situada en el centro.
- El planeta Web, que es una esfera de radio 1, trasladada a las coordenadas 10.0 6.0 10.0.
- El planeta Chat, que es una esfera de radio 1, trasladada a las coordenadas 25.0 1.0 -7.0.
- El planeta Web3D que es un cubo de lado 1, trasladado a las coordenadas -8.0 8.0 27.0.
- Un letrero de bienvenida, trasladado a las coordenadas -7.0 9.0 25.0.

El código del documento Progra10.wrl es:

```
#VRML V2.0 utf8
#Progra10.wrl
#Aplicación del nodo Transform
Shape {
    #Campo appearance:
    appearance DEF Fondo Appearance {
        material Material {   }
    }
    #Campo geometry:
    geometry Sphere {
        radius 10
    }
}
Transform {
    translation 10.0 6.0 10.0
    children [
        Shape{
            appearance USE Fondo
            geometry Sphere {
                radius 1
            }
        },
    ]
}
Transform {
    translation 25.0 1.0 –7.0
    children [
```

Figura 3.13. Ejemplo Práctico

```
            Shape{
                appearance USE Fondo
                geometry Sphere {
                    radius 1
                }
            },
        ]
    }
    Transform {
        translation -0.8 8.0 27.0
        children [
            Shape{
                appearance USE Fondo
                geometry Box {
                    size 1 1 1
                }
            },
        ]
    }
    Transform {
        translation -1.8 -8.0 20.0
        children [
            Shape{
                appearance USE Fondo
                geometry Cone {
                    height 1.5
                    bottomRadius 1.0
                }
            },
        ]
    }
    Transform {
        translation -17.0 14.0 25.0
        children [
            Shape{
                appearance USE Fondo
                geometry Text {
                    string ["BIENVENIDOS",
                    "A NUESTRA GALAXIA VIRTUAL"]
                    fontStyle FontStyle {
                        family "ARIAL",
                        style "BOTH",
                        size 5.0
                        spacing 1.0
                        justify "MIDDLE"
                    }
                }
            },
        ]
    }
    Transform {
        translation 7.0 1.0 12.0
        children [
            Shape{
                appearance USE Fondo
                geometry Text {
                    string ["VRML"]
                    fontStyle FontStyle {
                        family "ARIAL",
                        style "BOTH",
                        size 5.0
```

```
                    spacing 1.0
                    justify "MIDDLE"
                }
            }
        },
    ]
}
```

Los cuerpos todavía no tienen ninguna textura o color. Además el punto de vista inicial es el que hay por defecto. Por tanto, puede ocurrir que el visor se sitúe inicialmente en un sitio inapropiado (como es el caso del Cosmo Player, que se sitúa dentro de la esfera mayor). En este caso, será necesario alejarse, utilizando el control Zoom. Mas adelante se verá cómo situarlo en el lugar correspondiente.

3.13.6. Nodo Switch

EL nodo Switch (bascular o conmutar*)* sirve para agrupar otros nodos, pero con la diferencia de que sólo será utilizado uno de los hijos agrupados. Su estructura es la siguiente:

```
Switch {
    whichChoice 0
     choice [...]
    choice [...]
    choice [...]
}
```

Como se puede ver, este nodo tiene un campo whichChoice (cuál elección) cuyo valor es un número que indica cual hijo choice es el elegido (0, 1, 2, etc.). Si este valor es -1, indica que no está elegido ninguno.

Un posible uso de este nodo es el de tener preparadas diferentes versiones del nodo Shape, es decir, diferentes formas de un objeto. Bastará con variar el valor de whichChoice para pasar rápidamente de una forma a otra. Esta variación se hará utilizando los eventos.

3.13.7. Nodo Billboard

El nodo Billboard (cartel) sirve para agrupar otros nodos hijos. Todos los nodos agrupados serán visibles al contrario de nodo Switch, en el que sólo es visible uno de ellos.

La particularidad de este nodo es que crea un sistema de coordenadas para los nodos agrupados que están siempre de cara al espectador.

Su estructura es la siguiente:

```
Billboard {
    axisOfRotation eje_x eje_y eje_z
    children [ . . . ]
}
```

Tiene un campo, axisOfRotation (eje de rotación), cuyo valor es un grupo de tres cifras que indican cuál es el eje alrededor del que se efectúa automáticamente el giro para permanecer siempre de cara. La convención que se utiliza es la misma que la vista para la rotación, excepto que en este caso no hace falta poner un cuarto valor para expresar el ángulo:

- Rotación alrededor del eje X: 1 0 0.
- Rotación alrededor del eje Y: 0 1 0.
- Rotación alrededor del eje Z: 0 0 1.

A continuación en el ejemplo Progra11.wrl se ve como utilizar este nodo:

```
#VRML V2.0 utf8
#Progra11.wrl
#Ejemplo practico del nodo Billboard
Billboard {
    axisOfRotation 0 1 0
    children Shape {
        geometry IndexedFaceSet {
            coord Coordinate {
                point [ 2 3 0, -2 3 0, -2 -3 0, 2 -3 0 ]
            }
            coordIndex [ 0 1 2 3 ]
            solid FALSE
            texCoord TextureCoordinate {
            point [ 1 1, 0 1, 0 0, 1 0 ]
            }
        }
        appearance Appearance {
            texture ImageTexture {
                url "rabbit.gif"
            }
        }
    }
}
```

Figura 3.14. Ejemplo del Nodo Billboard

3.13.8. Nodo Anchor

Este nodo, permite hacer enlaces (o hiperenlaces) a otros archivos en la Web. Su sintaxis es:

```
Anchor{
    description " Descripcion_del_enlace"
    parameter [ ... ]
    url [ dirección_URL]
    bboxCenter 0 0 0
    bboxSize -1 -1 -1
    children [ ... ]
}
```

En el campo children están los hijos del nodo. Cuando el usuario elige uno de ellos, el navegador salta a la dirección especificada en el campo *url*. La forma en que el usuario elige el objeto depende del navegador, pero normalmente será mediante un click de ratón sobre su geometría.

En el campo description, el implementador puede escribir una cadena que se mostrará al usuario cuando el ratón pase por encima de los objetos hijos.

En el campo url se indicará el URL o URL'S (por orden de preferencia) a los que el navegador deberá saltar si el usuario elige algún objeto hijo. Si el fichero contiene un escenario VRML, este reemplazará al escenario actual. Si es otro tipo de documento, el navegador decidirá la forma de mostrarlo.

Para entender el manejo de este nodo tiene el ejemplo Progra12.wrl:

```
#VRML V2.0 utf8
#Progra12.wrl
#Utilización de Nodo Anchor
Transform {
    center 1 0 0
    children [
        Anchor {
            description "Una lata"
            url ["lata.wrl"]
            children [
                Shape {
                    appearance Appearance {
                        material Material {
                            diffuseColor 1 1 0
                        }
                    }
                    geometry Text {
                        string ["Un Escenario VRML"]
                        fontStyle FontStyle {
                            family ["Tahoma", "SANS"]
                            size 2.0
                            justify ["CENTER"]
                        }
                    }
                }
            ]
        }
    ]
}
```

Figura 3.15.1. Ejemplo del Nodo Anchor

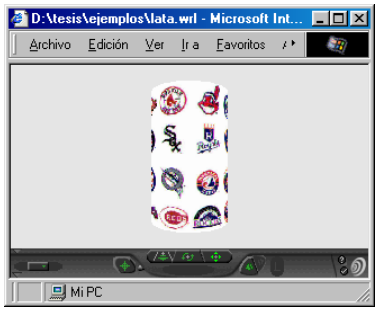


Figura 3.15.2. Ejemplo del Nodo Anchor

3.13.9. Nodo Collision

Por defecto, en VRML no es posible atravesar los objetos que componen una escena (es decir, el usuario no se puede comportar como un fantasma).

Sin embargo, este nodo permite desactivar las colisiones contra los objetos hijos, de modo que el usuario pueda atravesarlos. También permite definir un objeto más simple contra el que se va a colisionar (un proxy) para acelerar la detección de colisiones. Su sintaxis es:

```
Collision{
    children [ ... ]
    collide valor_lógico
    proxy nodo
}
```

children	Están los nodos a los que se desea cambiar la forma de colisionar.
collide	Es un booleano que indica si se computarán las colisiones (TRUE) o estarán desactivadas para los nodos hijos (FALSE) permitiendo al usuario atravesarlos.
proxy	Contiene el objeto u objetos alternativos contra los que se chocará en vez de los hijos. Estos objetos serán invisibles.

Por defecto, el campo collide toma el valor TRUE. Si no se especifica ningún objeto hijo y se especifica un proxy, el resultado será que el usuario chocará contra un objeto invisible (obviamente con la geometría del *proxy*).

Un ejemplo práctico de como utilizar el nodo Collision tiene en el ejemplo Progra13.wrl:

```
#VRML V2.0 utf8
#Progra13.wrl
#Ejemplo practico de Collision
Collision {
    collide FALSE
    children [
        Transform {
            translation -3 0 0
            children [
                Shape {
                    geometry Box {
                        size 2 2 2
                    }
                    appearance Appearance {
                        material Material {
                            diffuseColor 1 0 0
                        }
                    }
                }
            ]
        }
    ]
}
Collision {
    collide TRUE
    children [
        Shape {
            geometry Sphere {
                radius 1
            }
            appearance Appearance {
```

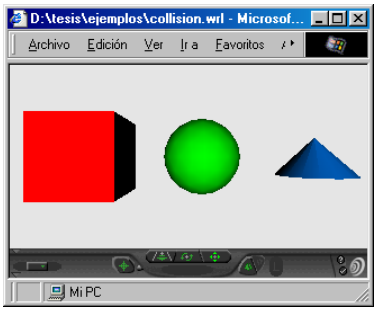


Figura 3.16. Ejemplo del Nodo Collision

```
                material Material {
                    diffuseColor 0 1 0
                }
            }
        }
        Transform {
            translation 3 0 0
            children [
                Shape {
                    geometry Cone {
                        height 1
                        bottomRadius 1
                    }
                    appearance Appearance {
                        material Material {
                            diffuseColor 0 0.5 1
                        }
                    }
                }
            ]
        }
    ]
}
```

3.14. FORMAS COMPLEJAS

Hasta ahora, para crear los distintos objetos que forman un escenario de realidad virtual, se han utilizado exclusivamente los nodos primitivos (caja, cono, cilindro y esfera).

Tal como se ha visto anteriormente, agrupando y modificando estas formas primitivas se pueden conseguir otras formas más complejas.

Pero esta técnica tiene un límite. Supóngase que se quiere representar un automóvil, por ejemplo. Con la sola utilización de las formas primitivas sería demasiado complicado de crear, y se generarían ficheros VRML demasiado voluminosos.

Hay un método más eficaz para construir las formas complejas, consiste en definir una serie de puntos en el espacio y luego unirlos entre sí para crear líneas o superficies.

3.14.1. Nodo Coordinate

Este nodo sirve para definir una serie de puntos en el espacio tridimensional y no para representarlos, sino que se utilizan dentro de otros nodos, como se ve a continuación.

La estructura de este nodo es la siguiente:

```
Coordinate {
    point [
        2 11 17.1,
        20.5 13.8 5.3,14 6.1 22.9
    ]
}
```

Tiene un campo point, cuyo valor son grupos de tres cifras, separadas por comas, englobadas entre corchetes [y].

Cada grupo de tres cifras representa un punto en el sistema de coordenadas (como se explico en nodo Transform). En el ejemplo están definidos tres puntos, pero se pueden poner tantos como se quiera. Este nodo se utiliza dentro de otros nodos.

3.14.2. Nodo PointSet

Sirve para crear puntos aislados. Su estructura es:

```
PointSet {
    coord Coordinate {
        point [ . . . ]
    }
    color Color {
        color [ . . . ]
    }
}
```

La descripción de sus parámetros son los siguientes:

coord	Su valor es el nodo Coordinate visto anteriormente
color	Tiene como valor el nodo Color, que a su vez tiene como valor el campo color de nuevo.

Como se ha explicado, el nodo Shape tiene dos campos: appearance que define la apariencia del objeto (color, textura, etc.) y con el campo geometry su forma. Por tanto, es evidente que la inclusión del nodo PointSet deberá hacerse en este último campo, ya que la geometría de este objeto es la de un grupo de puntos. De manera resumida, quedará de esta manera:

```
Shape {
    geometry PointSet { ... }
}
```

Desarrollando el ejemplo Progra14.wrl su código es:

```
#VRML V2.0 utf8
#Progra14.wrl
#Ejemplo de un grupo de tres puntos con colores
```

```
Shape {
    geometry PointSet {
        coord Coordinate {
            point [
                12 11 17.1,
                20.5 13.8 5.3,
                14 6.1 22.9
            ]
        }
        color Color {
            color [
                1 0 0,    # 1º punto rojo
                0 1 1,    # 2º punto verde
                1 1 0     # 3º punto amarillo
            ]
        }
    }
}
```

Obsérvese que se ha prescindido del campo appearance del nodo Shape, ya que no es necesario, pues los puntos no van a tener la apariencia por defecto, sino la que se determina en el campo color.

En el campo color se puede observar que hay (entre corchetes) tres grupos de números, separados por comas. Cada grupo de tres cifras representa un color, para cada punto expresado en el campo point, y correspondiéndose en el mismo orden.

3.14.3. Ejemplo práctico con el nodo PointSet y Coordinate

Como ejercicio práctico de este nodo se definirá tres grupos de 10 puntos, el primero de ellos de color verde, el segundo de color amarillo y el tercero de color rojo con los puntos definidos a continuación:

P. verdes	1 1 0	2 1 0	3 1 0	4 1 0	5 1 0	6 1 0	7 1 0	8 1 0	9 1 0	10 1 0
P. amarrillos	1 2 0	2 2 0	3 2 0	4 2 0	5 2 0	6 2 0	7 2 0	8 2 0	9 2 0	10 2 0
P. rojos	1 3 0	2 3 0	3 3 0	4 3 0	5 3 0	6 3 0	7 3 0	8 3 0	9 3 0	10 3 0

Desarrollo:

```
#VRML V2.0 utf8
#Progra15.wrl
#Ejemplo de un grupo de puntos
Shape {
    geometry PointSet {
        coord Coordinate {
            point [
                1 1 0, 2 1 0, 3 1 0, 4 1 0, 5 1 0,
                6 1 0, 7 1 0, 8 1 0, 9 1 0, 10 1 0,
```

```
                1 2 0, 2 2 0, 3 2 0, 4 2 0, 5 2 0,
                6 2 0, 7 2 0, 8 2 0, 9 2 0, 10 2 0,
                1 3 0, 2 3 0, 3 3 0, 4 3 0, 5 3 0,
                6 3 0, 7 3 0, 8 3 0, 9 3 0, 10 3 0,
            ]
        }
        color Color {
            color [
                0 1 1, 0 1 1, 0 1 1, 0 1 1, 0 1 1,  #punto verde
                0 1 1, 0 1 1, 0 1 1, 0 1 1, 0 1 1, 0 1 1,
                1 1 0, 1 1 0, 1 1 0, 1 1 0, 1 1 0, #punto amarillo
                1 1 0, 1 1 0, 1 1 0, 1 1 0, 1 1 0, 1 1 0,
                1 0 1 , 1 0 1, 1 0 1, 1 0 1, 1 0 1, #punto rojo
                1 0 1, 1 0 1, 1 0 1, 1 0 1, 1 0 1, 1 0 1

            ]
        }
    }
}
```

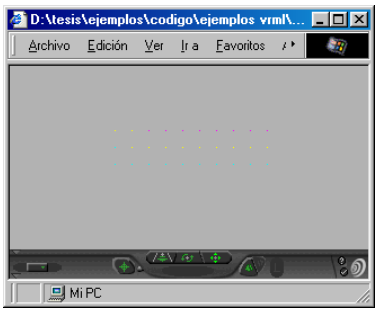


Figura 3.17. Aplicación utilizando los Nodos PointSet y Coordinate

3.14.4. Nodo IndexedLineSet

Con este nodo se crean líneas, uniendo una serie de puntos determinados. Su estructura es la siguiente:

```
IndexedLineSet {
    coord Coordinate {
        point [ . . . ]
    }
    coordIndex [ . . . ]
}
```

A continuación se describe la utilidad de cada campo:

coord	Su valor es el nodo Coordinate ya explicado, el cual, sirve para determinar las coordenadas de los puntos aislados que se van a unir entre sí para formar las líneas.
coordIndex	Con este campo se define el orden con el que se unen los puntos entre sí, dónde empieza y acaba una línea, etc., como se verá a continuación.
color	Este campo especifica cuántos y qué colores se van a usar (en el caso de que haya más de una línea).
colorIndex	Este campo atribuye a cada una de las líneas existentes alguno de los colores expresados en el campo anterior.
colorPerVertex	Especifica si los colores serán colores continuos, o un gradiente entre dos colores.

Para que un nodo pueda verse, debe ser parte del nodo Shape. Por tanto, la inclusión del nodo IndexedLineSet en el nodo Shape se visualiza en el ejemplo Progra16.wrl:

```
#VRML V2.0 utf8
#Progra16.wrl
#Ejemplo de línea uniendo tres puntos, sin definición del color
Shape {
    geometry IndexedLineSet {
        coord Coordinate {
            point [
                5 5 0,   #este es el punto 0
                15 9 0,  #este es el punto 1
                20 18 0  #este es el punto 2
            ]
        }
        coordIndex [
            0, 1, 2, -1
        ]
    }
}
```

En el campo coord se establecen las coordenadas de los tres puntos (tal como se explico en el nodo Coordinate). A cada coordenada se ha añadido una línea de comentario para indicar cuál es la numeración que le corresponde al punto. Obsérvese que se comienza por 0 (y no por 1).

En el campo coordIndex se establece el orden en el que se unen los tres puntos. En este caso la línea empieza en el primero (0), sigue con el segundo (1) y acaba en el tercero (2). Con el valor -1 se indica que ahí acaba la línea. Se pueden unir los puntos en el orden que se desee, como por ejemplo (0,2,1), resultando una línea distinta. Habría que modificar el campo coordIndex, que quedaría de esta manera:

```
coordIndex [
    0, 2, 1, -1
]
```

Se pueden crear simultáneamente diversas líneas. Supóngase que en el campo point se han especificado las coordenadas de 12 puntos (por tanto, del 0 al 11), y se desea crear dos líneas, la primera con los 6 primeros puntos y la segunda con los 6 últimos. El campo coordIndex quedaría de esta manera:

```
coordIndex [
    0, 1, 2, 3, 4, 5, -1,    #esta es una línea
    6, 7, 8, 9, 10, 11, -1   #esta es otra línea
]
```

Un ejemplo completo con este nodo esta en el Progra17.wrl, adicionalmente se le dará un color verde a la línea.

```
#VRML V2.0 utf8
#Progra17.wrl
#Ejemplo de línea de color verde, uniendo tres puntos
Shape {
    geometry IndexedLineSet {
        coord Coordinate {
            point [
                5 5 0,   #este es el punto 0
                15 9 0,  #este es el punto 1
                20 18 0  #este es el punto 2
            ]
        }
        coordIndex [
            0, 1, 2, -1
        ]
        color Color {
            color [
                0 1 0   #este es el color 0 (verde),
            ]           #el único posible pues solo hay una línea
        }
        colorIndex [
            0   #a la única línea se le atribuye el único color 0
        ]
        colorPerVertex FALSE
    }
}
```

Figura 3.18. Utilización del Nodo IndexedLineSet

3.14.5. Ejemplo práctico con el nodo IndexedLineSet

Creé un grupo de 10 puntos verdes y otro de 10 puntos morados con las coordenadas siguientes:

P. verdes	1 1 0	2 4 0	3 5 0	4 4 0	5 6 0	6 7 0	7 5 0	8 6 0	9 4 0	10 3 0
P. morados	1 3 0	2 2 0	3 2 0	4 1 0	5 2 0	6 4 0	7 3 0	8 5 0	9 5 0	10 6 0

En el campo point se pondrán las coordenadas de los 20 puntos, que tienen una numeración del 0 al 19.

En el campo coordIndex se expresa que existen dos líneas: la primera uniendo ordenadamente los puntos 0 al 9, y la segunda del 10 al 19. Por tanto quedará así:

```
coordIndex [
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9, -1,   # primera línea
    10, 11, 12, 13, 14, 15, 16, 17, 18, 19, -1   #segunda línea
]
```

Como son dos colores, el campo color deberá ser:

```
color [
    0 1 0   # color 0 (verde)
    1 0 1   # color 1 (morado)
]
```

En el campo colorIndex se expresará que el orden de los colores atribuidos es el 0 (verde) para la primera línea y el 1 (morado), para la segunda:

```
colorIndex [
    0, 1
]
```

Desarrollo:

```
#VRML V2.0 utf8
#Progra18.wrl
#Ejemplo de dos líneas de color verde y morado, uniendo tres puntos
Shape {
    geometry IndexedLineSet {
        coord Coordinate {
            point [
                1 1 0, 2 4 0, 3 5 0, 4 4 0, 5 6 0,
                6 7 0, 7 5 0, 8 6 0, 9 4 0, 10 3 0,
                1 3 0, 2 2 0, 3 2 0, 4 1 0, 5 2 0,
                6 4 0, 7 3 0, 8 5 0, 9 5 0, 10 6 0
            ]
        }
        coordIndex [
            0, 1, 2,3,4,5,6,7,8,9 -1
            10,11,12,13,14,15,16,17,18,19, -1
        ]
        color Color {
            color [
                0 1 0,   #este es el color 0 (verde),
                1 0 1     #color rojo
            ]
        }
        colorIndex [
            0,   #color 0 de la línea 0
            1
        ]
        colorPerVertex FALSE
    }
}
```

Figura 3.19. Trazo de líneas en el Escenario Virtual

3.14.6. Nodo IndexedFaceSet

Este nodo sirve para obtener caras (superficies planas) definidas por varios puntos.

La idea es unir una serie de puntos para que formen una polilínea cerrada, que es la que definirá la cara.

Su estructura es:

```
IndexedFaceSet{
    coord Coordinate {
        point [ . . . ]
    }
    coordIndex [...]
    color Color {
        color [ . . . ]
    }
    colorIndex [...]
    colorPerVertex ...
}
```

Estos son los campos que tiene este nodo, los mismos que se describen a continuación:

coord	Su valor es el *nodo Coordinate*, sirve para determinar las coordenadas de los puntos aislados que se van a unir entre sí para formar las caras.
coordIndex	Este campo define el orden con el que se unen los puntos entre sí para formar las distintas caras
color	Especifica cuántos y qué colores se van a usar (en el caso de que haya más de una cara).
colorIndex	Sirve para atribuir a cada una de las caras existentes alguno de los colores expresados en el campo color.
colorPerVertex	Especifica si los colores serán colores continuos, o un gradiente entre dos colores.

Para que un nodo pueda verse, debe estar incrustado en el nodo Shape. Por tanto, la inclusión del nodo IndexedFaceSet en el nodo Shape se hará de esta manera:

```
Shape {
    geometry IndexedFaceSet { ... }
}
```

El código del Progra19.wrl es un ejemplo sencillo de tres puntos que se unen ordenadamente entre sí para formar una cara.

```
#VRML V2.0 utf8
#Progra19.wrl
#Ejemplo de cara uniendo tres puntos,
#sin definición del color
Shape {
    geometry IndexedFaceSet {
        coord Coordinate {
            point [
                5 5 0,   #este es el punto 0
                15 9 0,  #este es el punto 1
                20 18 0  #este es el punto 2
            ]
        }
        coordIndex [
```

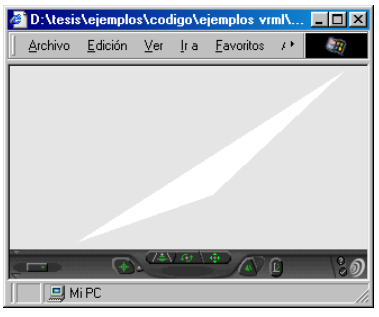


Figura 3.20. Ejemplo de un plano

```
            0, 1, 2, -1
        ]
    }
}
```

En el campo coord se establecen las coordenadas de los tres puntos. A cada coordenada se ha añadido una línea de comentario para indicar cuál es la numeración que le corresponde al punto. Obsérvese que se comienza por 0 (y no por 1).

En el campo coordIndex se establece el orden en el que se unen los tres puntos que van a formar la polilínea cerrada que va a formar la cara. Con el valor -1 se indica que ahí acaba la cara.

Se puede observar que se ha creado una cara triangular, sin color ni apariencia definida.

Se pueden definir más de una cara simultáneamente, como se verá más adelante.

En el programa Progra20.wrl se pone de color verde la cara del ejemplo anterior o Progra19.wrl.

```
#VRML V2.0 utf8
#Progra20.wrl
#Ejemplo de cara, uniendo tres puntos, de color verde
Shape {
    geometry IndexedFaceSet {
        coord Coordinate {
            point [
                5 5 0,   #este es el punto 0
                15 9 0,  #este es el punto 1
                20 18 0  #este es el punto 2
            ]
        }
        coordIndex [
            0, 1, 2, -1
        ]
        color Color {
            color [
                0 1 0   #este es el color 0 (verde),
            ]
        }
        colorIndex [
            0   #a la única cara se le atribuye el único color 0
        ]
        colorPerVertex FALSE
    }
}
```

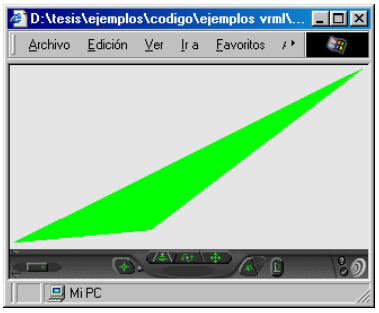


Figura 3.21. Aplicando color al plano

En el campo color se especifica el color (o los colores, sí existe más de una cara) y en el campo colorIndex se atribuye el color a la cara (o caras, sí existe más de una).

Si se manipula el escenario, se puede observar que la cara verde es visible sólo por un lado, el que está de frente.

3.14.6.1. Determinando una cara visible

Por defecto, la cara visible es la perpendicular a la parte positiva de los ejes coordenados. En el ejemplo anterior, la cara es perpendicular a la parte positiva del eje Z.

Pero puede haber ocasiones que interese que sea visible la otra cara. Para determinar una cosa o la otra, el nodo IndexedFaceSet puede tener el campo **ccw** (abreviatura de counterclockwise: contrario a las agujas del reloj). Existen dos posibilidades:

- **ccw TRUE** es la opción por defecto, por tanto no es obligatoria ponerlo. La cara visible es la perpendicular a la parte positiva de los ejes coordenados.
- **ccw FALSE** en este caso, la cara visible es la opuesta a la parte positiva de los ejes coordenados

Si en el ejemplo anterior se añade el campo **ccw FALSE**. Para poder ver la cara verde hay que girar 180º el escenario.

3.14.6.2. Determinando ambas caras visibles

Se puede hacer que ambas caras sean visibles simultáneamente. Para ello el nodo IndexedFaceSet puede tener el campo **solid**, que tiene dos posibilidades:

solid TRUE es la opción por defecto, sólo una cara es visible la determinada por el campo ccw, y por tanto no es obligatorio ponerlo.

solid FALSE en este caso, ambas caras son visibles simultáneamente.

Si al ejemplo anterior de la cara verde, se le añade el campo **solid FALSE**. Girando el escenario, se verá que ambas caras son visibles.

3.14.6.3. Utilización de este nodo para crear un suelo

Un uso muy útil del nodo IndexedFaceSet es el de crear un suelo para un escenario de realidad virtual.

Para crear un suelo de color azul para un escenario, de dimensiones 100x100. Las coordenadas de las cuatro esquinas serán:

50,0,50 -50,0,50 -50,0,-50 50,0,-50

Añada el campo **solid FALSE**, visto. Pero al cargar el escenario, en principio no se verá nada. Esto es debido a que el punto de vista por defecto está precisamente a nivel cero, es decir, en el mismo suelo. Sólo se verá el suelo si se hace girar todo el escenario.

Con este método se obtendrá un suelo del color que se desee, pero se pueden obtener suelos que tengan una textura determinada (hierba, piedra, etc.), esto se verá más adelante.

Con el nodo IndexedFaceSet se crean suelos planos. Después se verá cómo crear suelos accidentados (terrenos con distintas elevaciones, colinas, etc.).

3.14.7. Ejercicio práctico utilizando el nodo IndexedFaceSet

Partiendo de estos cuatro puntos, de coordenadas:

5 5 0, # punto 0 15 9 0, # punto 1 20 18 0, # punto 2 20 0 15, # punto 3

Se propone crear tres caras contiguas, visibles por ambos lados y de distinto color, de la siguiente manera:

- Una cara verde, con los puntos 0 -1 -2.
- Una cara amarilla, con los puntos 1 -2 -3.
- Una cara azul, con los puntos 0 -1 -3.

En el campo point se pondrán las coordenadas de los 4 puntos, que tienen una numeración del 0 al 3.

En el campo coordIndex se expresará que existen tres caras formadas por sus respectivos puntos. Por tanto quedará así:

```
coordIndex [
      0, 1, 2, -1,   # primera cara
      1, 2, 3, -1,   # segunda cara
      0, 1, 3, -1    #tercera cara
]
```

Como deben existir tres colores, el campo color deberá ser:

```
color [
      0 1 0,   # color 0 (verde)
      1 1 0,   # color 1 (amarillo)
      0 0 1    # color 2 (azul)
]
```

En el campo colorIndex se expresará que el orden de los colores atribuidos es el 0 (verde) para la primera cara, el 1 (amarillo), para la segunda y 2 (azul), para la tercera

```
colorIndex [
      0, 1, 2
]
```

Además, se añadirán los campos colorPerVertex FALSE y solid FALSE (para que sean visibles por ambos lados).

En el ejemplo Progra21.wrl, esta el resultado y su código:

```
#VRML V2.0 utf8
#Progra21.wrl
#Ejemplo de tres caras de diferente color y visibles por ambos lados
Shape {
    geometry IndexedFaceSet {
        coord Coordinate {
            point [
                5 5 0,
                15 9 0,
                20 18 0,
                20 0 15
            ]
        }
        coordIndex [
            0, 1, 2, -1,    #Cara 0
            1, 2, 3, -1,    #Cara 1
            0, 1, 3, -1     #Cara 2
        ]
        color Color {
            color [
                0 1 0,   #color 0 (verde)
                1 1 0,   #color 1 (amarillo)
                0 0 1    #Color 2 (azul)
            ]
        }
        colorIndex [
            0, 1, 2   #asignación de colores a las caras
        ]
        colorPerVertex FALSE
        solid FALSE
    }
}
```

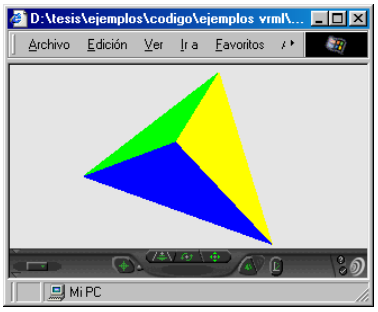


Figura 3.22. Triángulo formado con planos

3.14.8. Nodo ElevationGrid

Haciendo uso del nodo IndexedFaceSet se ha visto cómo crear un suelo plano para un escenario virtual. Ahora se verá cómo crear suelos accidentados, con distintas elevaciones, muy apto para crear cordilleras, fondos marinos, superficies de planetas, etc.

El nodo ElevationGrid (rejilla de elevaciones) determina una rejilla de puntos, es decir, una cuadrícula de puntos de separación uniforme en el plano horizontal XZ, a los que se atribuyen distintas cotas de altura. Se generará una superficie irregular uniendo dichas cotas. Su estructura general es:

```
ElevationGrid{
    xDimension    ...
    xSpacing      ...
    zDimension    ...
    zSpacing      ...
    height        ...
    color Color[...]
    colorPerVertext
}
```

Descripción de los campos:

xdimensiones	Número de puntos de la rejilla, en la dirección del eje X
xspacing	Separación de los puntos de la rejilla, en la dirección del eje X
zdimension	Número de puntos de la rejilla, en la dirección del eje Z
zspacing	Separación de los puntos de la rejilla, en la dirección del eje Z
height	Altura de cada uno de los puntos de la rejilla
color	Contiene el nodo Color, que a su vez contiene el campo color, en donde se especifican los colores atribuidos a cada cuadrícula.
colorPerVertex	Puede ser TRUE, para colores en gradiente. O puede ser FALSE, en cuyo caso los colores son continuos.

La rejilla debe comenzar siempre en el origen de coordenadas y crecer en el sentido positivo de los ejes X y Z. Como se ha dicho repetidas veces, para que un nodo que define una forma (como es el caso del nodo ElevationGrid) sea visible, debe incluirse dentro del nodo Shape

El nodo ElevationGrid será el valor del campo geometry, quedando por tanto:

```
Shape {
        appearance ...
        geometry ElevationGrid { ... }
}
```

Desarrollando el ejemplo Progra22.wrl con apariencia por defecto se tiene:

```
#VRML V2.0 utf8
#Progra22.wrl
#Ejemplo del nodo ElevationGrid,
#con la apariencia por defecto
Shape {
    appearance Appearance {
        material Material { }
    }
    geometry ElevationGrid {
        xDimension 6
        xSpacing 4
        zDimension 3
        zSpacing 8
        height [
            0.25, 4, 7, 3, 1.5, 0.25,
            1.5, 2.5, 1.5, 2.1, 1, 0,
            0.3, 0.3, 0, -2.7, -1.5, -3.7
        ]
    }
}
```

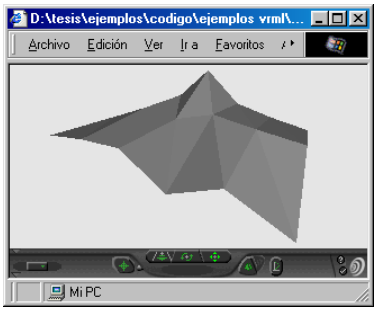


Figura 3.23. Ejemplo de ElevationGrid

En xDimension se tiene 6 puntos de la rejilla, en la dirección del eje X.

En xSpacing 4 de separación de los puntos de la rejilla, en la dirección del eje X.

En zDimension se tiene 3 puntos de la rejilla, en la dirección del eje Z.

En zSpacing 8 de separación de los puntos de la rejilla, en la dirección del eje.

En height que es la altura de cada uno de los puntos de la rejilla. En este caso, la rejilla tiene 6x3=18 puntos, por lo tanto, hay que poner las 18 cotas de altura que correspondan a cada punto.

Obsérvese que la superficie generada es sólida y existe gravedad, sí se camina por la superficie se "escalará" por las elevaciones.

Si se gira completamente el escenario, se puede comprobar también que la superficie sólo es visible por su parte delantera.

Pero en este caso concreto, hay una manera de determinar el color de partes determinadas de la superficie, como se verá a continuación.

En este caso, la rejilla está formada por dos hileras de 5 cuadrículas, es decir, en total 10 cuadrículas. Se puede determinar el color de la superficie generada sobre cada cuadrícula.

Se tiene la representación de la rejilla vista desde arriba. Además, se ha escrito en cada cuadrícula el color que se desea para la parte de la superficie que le corresponda y un número (del 1 al 10), para señalar el orden de atribución de colores.

Se ha escogido el color blanco para las zonas más altas, el rojo para las intermedias, el verde para las bajas y el azul para las más bajas.

```
#VRML V2.0 utf8
#Progra23.wrl
#Ejemplo del nodo ElevationGrid,
#con la apariencia a colores
Shape {
    geometry ElevationGrid {
        xDimension 6
        xSpacing 4
        zDimension 3
        zSpacing 8
        height [
            0.25, 4, 7, 3, 1.5, 0.25,
            1.5, 2.5, 1.5, 2.1, 1, 0,
            0.3, 0.3, 0, -2.7, -1.5, -3.7
        ]
        color Color {
            color [
                1 0 0   #rojo para la cuadr. 1
                1 1 1   #blanco para la cuadr. 2
                1 1 1   #blanco para la cuadr. 3
                1 0 0   #rojo para la cuadr. 4
                0 1 0   #verde para la cuadr. 5
                0 1 0   #verde para la cuadr. 6
                0 1 0   #verde para la cuadr. 7
                0 1 0   #verde para la cuadr. 8
                0 1 0   #verde para la cuadr. 9
                0 0 1   #azul para la cuadr. 10
            ]
        }
        colorPerVertex FALSE   #colores continuos (no degradados)
        solid FALSE   #ambas caras visibles
    }
}
```

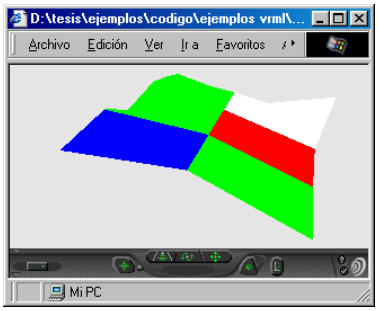


Figura 3.24. Aplicando color a ElevationGrid

El nodo ElevationGrid puede utilizar los campos **ccw** y **solid** los mismos que funcionan de acuerdo a lo explicado en el nodo IndexedFaceSet

3.14.9. Ejercicio práctico utilizando el nodo ElevationGrid

Se creará la superficie inversa a la del ejemplo práctico visto en la explicación de este nodo, es decir, mantener la misma rejilla y la misma distribución de colores, pero con las 18 cotas cambiadas de signo:

```
#VRML V2.0 utf8
#Progra24.wrl
#Ejemplo del nodo ElevationGrid,
#con colores
Shape {
    geometry ElevationGrid {
        xDimension 6
        xSpacing 4
        zDimension 3
        zSpacing 8
        height [
            -0.25, -4, -7, -3, -1.5, -0.25,
            -1.5, -2.5, -1.5, -2.1, -1, 0,
            -0.3, -0.3, 0, 2.7, 1.5, 3.7
        ]
        color Color {
            color [
                1 0 0   #rojo para la cuadr. 1
                1 1 1   #blanco para la cuadr. 2
                1 1 1   #blanco para la cuadr. 3
                1 0 0   #rojo para la cuadr. 4
                0 1 0   #verde para la cuadr. 5
                0 1 0   #verde para la cuadr. 6
                0 1 0   #verde para la cuadr. 7
                0 1 0   #verde para la cuadr. 8
                0 1 0   #verde para la cuadr. 9
                0 0 1   #azul para la cuadr. 10
            ]
        }
        colorPerVertex FALSE   #colores continuos (no degradados)
        solid FALSE   #ambas caras visibles
    }
}
```

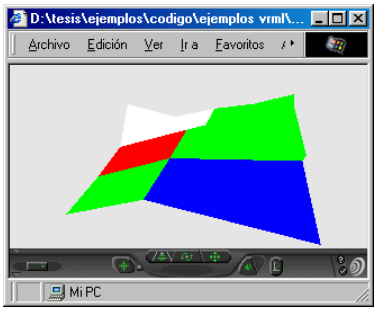


Figura 3.25. Elevación inversa del ejemplo anterior

3.15. COLOR Y TRANSPARENCIAS.

Anteriormente se ha comentado que el nodo Shape contiene dos campos, appearance y geometry, de los cuales el segundo indicaba el tipo de objeto a representar del que se ha hablado ampliamente. Sin embargo la misión del campo appearance apenas se ha comentado, ahora se procede a analizarlo con más detalle en este punto.

Como sé ha visto, para crear un objeto visible, se debe utilizar el nodo Shape, cuya estructura general es:

```
Shape {
    appearance ...
    geometry ...
}
```

El campo appearance se determina con el nodo Appearance:

```
Shape {
    appearance Appearance {
        material Material { }
    }
    geometry ...
}
```

El nodo Appearance tiene aquí un solo campo (material), con el que se va a determinar el color y grado de transparencia de los objetos. Pero en realidad puede tener otros campos, con los que se puede determinar su textura, como se verá a continuación.

3.15.1. Nodo Material

El campo appearance permitirá seleccionar el color y la textura del objeto que va a ser representado dentro del escenario virtual. Este campo toma como valor un nodo de tipo Appearance, el cual a su vez, posee un campo denominado material que toma como valor un nodo de tipo Material.

El nodo Material es el que controla las propiedades del color (selección del color, brillo, grado de transparencia, etc.) que se van a dar al objeto.

Los colores que se les dan a los objetos son colores RGB, es decir, vienen dados por tres valores en coma flotante, cada uno de los cuales representa uno de los colores primarios (Red (Rojo), Green(Verde), Blue(Azul)). El valor 0.0 representa la ausencia de color y el 1.0 la máxima intensidad.

Muchos programas de dibujo darán un valor para cada color RGB en un formato 255x255x255. Para poder utilizar estos colores en VRML es preciso convertirlos, dividiendo cada valor por 255 y colocando el resultado en su campo correspondiente dentro del nodo Material. Su sintaxis es:

```
Shape{
    appearance Appearance{
        material Material{
            diffuseColor        color_RGB
            emissiveColor       color_RGB
            specularColor       color_RGB
            ambientIntensity    valor_real
            transparency        valor_real
            shininess           valor_real
          }
    }
    geometry ...
}
```

Cada uno de los seis campos del nodo Material tiene su propio efecto específico sobre un objeto. Todos son opcionales.

diffuseColor	Este campo representa lo que la mayoría de los usuarios llamarían como el color del objeto.
emissiveColor	Se utiliza para fijar el color del brillo del objeto, cuando dicho objeto necesite ser visible en la oscuridad. De esta forma se consigue un efecto en donde la figura representada parece iluminada desde el interior mediante una luz de un determinado color. La configuración por defecto de este campo es el negro, ya que la mayoría de los objetos normalmente no brillan.
specularColor	Es un parámetro avanzado que permite indicar qué color de luz refleja el objeto. Por ejemplo, una cama roja no refleja un color rojo, pero una olla rojiza si puede reflejar su color.
ambientIntensity	Este campo es otro parámetro avanzado que indica la cantidad de luz ambiental (producida por los diferentes focos de luz del escenario virtual) es reflejada por el objeto. Toma valores en coma flotante entre 0.0 y 1.0.
shininess	Controla el brillo de un objeto. Toma valores en coma flotante entre 0.0 y 1.0.
transparency	Este campo indica el nivel de transparencia del objeto. Toma valores en coma flotante entre 0.0 y 1.0, siendo el 1.0 el nivel máximo de transparencia (objeto invisible) y el 0.0 el nivel mínimo (objeto totalmente opaco). El valor por defecto es el 0.0.

3.15.2. Código de colores

Un color se representa por un grupo de tres cifras. La primera cifra se refiere a la cantidad de color rojo, la segunda a la cantidad de color verde y la tercera a la cantidad de color azul.

He aquí algunos ejemplos de colores:

Color	**Rojo**	**Verde**	**Azul**
Rojo	1	0	0
Verde	0	1	0
Azul	0	0	1
Blanco	1	1	1
Negro	0	0	0
Amarillo	1	1	0
Violeta	1	0	1
Marrón	0.5	0.2	0

La siguiente aplicación tiene color externo e interno rojo y con una transparencia de 0.5:

```
#VRML V2.0 utf8
#Cono de color externo e interno rojo
Shape {
```

```
    appearance Appearance {
        material Material {
            diffuseColor 1 0 0
            emissiveColor 1 0 0
            transparency 0.5
        }
    }
    geometry Cone {
        height 3
        bottomRadius 0.75
    }
}
```

3.15.3. Ejercicio práctico utilizando el nodo Material

Como ejercicio práctico de este nodo, se pondrá colores a algunos de los objetos del ejercicio planteado en la sección 3.13.5.4 de la siguiente manera:

- Planeta VRML un color difuso de valor 1 0.5 1 (violeta oscuro) y un grado de transparencia de 0.5.
- Planeta Web un color difuso de valor 1 0.3 0.3 (marrón claro) y un grado de transparencia de 0.6.
- Letrero de bienvenida: color difuso amarillo (1 1 0).
- Los otros dos objetos (planeta Chat, planeta Web3D y el planeta Cono) se dejan de momento como están, ya que van a tener su propia textura.

```
#VRML V2.0 utf8
#Progra25.wrl
#Aplicación del nodo Transform
Shape {
    #Campo appearance:
    appearance Appearance {
        material Material {
            diffuseColor 1 0.5 1
            transparency 0.5
        }
    }
    #Campo geometry:
    geometry Sphere {
        radius 10
    }
}

Transform {
    translation 10.0 6.0 10.0
    children [
        Shape{
            appearance Appearance {
                material Material {
                    diffuseColor 1 0.3 0.3
                    transparency 0.6
```

Figura 3.26. Visualización de colores

```
                }
            }
            geometry Sphere {
                radius 1
            }
        },
    ]
}
Transform {
    translation 25.0 1.0 -7.0
    children [
        Shape{
            appearance DEF Fondo Appearance {
                material Material{}
            }
            geometry Sphere {
                radius 1
            }
        },
    ]
}
Transform {
    translation -0.8 8.0 27.0
    children [
        Shape{
            appearance USE Fondo
            geometry Box {
                size 1 1 1
            }
        },
    ]
}
Transform {
    translation -17.0 14.0 25.0
    children [
        Shape{
            appearance Appearance {
                material Material {
                    diffuseColor 1 1 0
                }
            }
            geometry Text {
                string ["BIENVENIDOS",
                "A NUESTRA GALAXIA VIRTUAL"]
                fontStyle FontStyle {
                    family "ARIAL",
                    style "BOTH",
                    size 5.0
                    spacing 1.0
                    justify "MIDDLE"
                }
            }
        },
    ]
}
Transform {
    translation 7 1 12.0
    children [
        Shape{
            appearance USE Fondo
            geometry Text {
```

```
                string ["VRML"]
                fontStyle FontStyle {
                    family "ARIAL",
                    style "BOTH",
                    size 4.0
                    spacing 1.0
                    justify "MIDDLE"
                }
            }
        },
    ]
}
Transform {
    translation -1.8 -8.0 20.0
    children [
        Shape{
            appearance USE Fondo
            geometry Cone {
                height 1.5
                bottomRadius 1.0
            }
        },
    ]
}
```

3.16. TEXTURA

Hasta ahora, para definir un objeto visible se ha utilizado el nodo Shape de la siguiente forma:

```
Shape {
    appearance Appearance {
        material ...
    }
    geometry ...
}
```

En donde el nodo Appearance tiene un solo campo, material, con el que se definen el color y la transparencia, como se observo anteriormente.

Pero en realidad puede tener también otros dos campos: texture y textureTransform, con los que se define la textura de los objetos:

```
Shape {
    appearance Appearance {
        material ...
        texture ...
        textureTransform ...
    }
    geometry ...
}
```

3.16.1. ¿Qué es la textura?

La textura es la posibilidad que se tiene de envolver un objeto con:

- Una imagen existente, usando el campo ImageTexture.
- Una película, usando el campo MovieTexture.
- Una imagen creada pixel a pixel, usando el campo PixelTexture.

A continuación se detallará cada uno de los campos.

3.16.2. Nodo ImageTexture

Esté indica una textura, referenciada por un URL, así como dos parámetros que dicen si la textura se ha de repetir vertical u horizontalmente a lo largo de todas las caras del objeto.

La sintaxis del nodo ImageTexture es la siguiente:

```
Shape {
    appearance Appearance {
        texture ImageTexture {
            url [ ... ]
            repeatS Valor_lógico
            repeatT  Valor_lógico
        }
    }
    geometry ...
}
```

url [ ]	Define la localización de la imagen. Los formatos aceptados son JPEG, GIF y PNG.
repeatS	Campo booleano que indica si la textura se tiene que repetir verticalmente.
repeatT	Campo booleano que indica si la textura se tiene que repetir horizontalmente.

Aplicando esto a una caja de dimensiones 1.5x2.2x0.5, se tendrá el siguiente resultado.

```
#VRML V2.0 utf8
# Ejemplo de caja con textura de una imagen
Shape {
    appearance Appearance {
        texture ImageTexture {
            url ["Utn.jpg "]
                repeatS FALSE
                repeatT FALSE
            }
        }
    geometry Box {
```

```
        size 1.5 2.2 0.5
    }
}
```

De tal forma que el objeto que se crea tendrá un gráfico como textura en todas sus caras.

3.16.2.1. Imágenes distintas en un mismo objeto

Para adicionar texturas a las diferentes caras de un objeto, se sigue un tratamiento diferente así, si se desea hacer una lata se realizaran las siguientes operaciones.

Se define el cilindro con la imagen etiqueta.jpg, pero se anulan las superficies superior (top) e inferior (bottom). A continuación se define otro cilindro idéntico con la imagen fondo.jpg, pero se anula la superficie lateral (side).

Por medio del nodo Group se agrupan ambos cilindros y el resultado es aparentemente un cilindro con imágenes distintas.

Realizando estas operaciones el resultado esta en el Progra26.wrl y es el siguiente:

```
#VRML V2.0 utf8
#Progra26.wrl
#Ejemplo del nodo ImageTexture
#imagen Lateral
Shape {
    appearance Appearance {
        texture ImageTexture {
            url ["etiqueta.jpg"]
        }
    }
    #Nodo Cylinder
    geometry Cylinder {
        height 2
        radius 0.6
        top FALSE
        bottom FALSE
    }
}
Shape {
    appearance Appearance {
        texture ImageTexture {
            url ["fondo.jpg"]
        }
    }
    geometry Cylinder {
        height 2
        radius 0.6
        side FALSE
    }
}
```

Figura 3.27. Visualización de una lata

3.16.3. Nodo MovieTexture

En lugar de usar imágenes estáticas como textura de los objetos, se pueden utilizar vídeos (películas), en formato MPEG, haciendo uso del nodo MovieTexture, en vez de ImageTexture, de la siguiente manera:

```
texture MovieTexture {
    url "mivideo.mpg"
    repeatS Valor_lógico
    repeatT Valor_lógico
    loop valor_lógico
    speed Valor
    startTime, stopTime
}
```

url [ ]	Define la localización de la película.
repeatS	Campo booleano que indica si la textura se tiene que repetir verticalmente.
repeatT	Campo booleano que indica si la textura se tiene que repetir horizontalmente.
loop	Controla si el vídeo funciona ininterrumpidamente (TRUE) o una sola vez (FALSE).
speed	Define la velocidad con la que se debe pasar la película. Por ejemplo, si este campo vale 2, la película se pasa al doble de velocidad. Si se asigna un valor negativo, la película se pasa en reversa.
startTime, stopTime	Indican, respectivamente, el momento de comienzo y de finalización de la película, en segundos transcurridos desde la fecha de creación del Universo en VRML.

Aplicando lo aprendido a la caja del ejemplo anterior se tiene:

```
#VRML V2.0 utf8
# Ejemplo de caja con textura de una imagen
Shape {
    appearance Appearance {
        texture MovieTexture {
            url ["swpro.mpg "]
            repeatS FALSE
            repeatT FALSE
            loop TRUE
            speed 1
        }
    }
    geometry Box {
        size 1.5 2.2 0.5
    }
}
```

3.16.4. Nodo PixelTexture.

Este nodo permite definir una textura pixel a pixel. Tiene tres campos: La sintaxis del nodo es:

```
PixelTexture {
    image 0 0 0
    repeatS Valor_lógico
    repeatT Valor_lógico
}
```

image Define la imagen que se utilizará como textura por medio de pixels. Los primeros tres valores de este campo definen respectivamente la anchura (en pixels), la altura (en pixels) y el número de bytes por pixel que se utilizarán. Los posibles valores de este último parámetro son:

1: Imagen en escala de grises.
2: Escala de grises con canal alfa para transparencia.
3: RGB.
4: RGB con canal alfa para transparencia.

repeatS Campo booleano que indica si la textura se tiene que repetir verticalmente.

repeatT Campo booleano que indica si la textura se tiene que repetir horizontalmente.

Los valores de color (o de gris), al especificarse en bytes, varían en este caso entre 0 y 255. Esto es diferente al modo en que se especifican el resto de los colores en VRML (valores entre 0 y 1).

Como se puede ver, por defecto este nodo no especifica ninguna textura. Aunque pueda parecer difícil especificar una textura por medio de pixels, se pueden lograr resultados interesantes con muy pocos valores. Por ejemplo:

```
#VRML V2.0 utf8
# Ejemplo de caja con textura de una imagen
Shape {
    appearance Appearance {
        texture PixelTexture {
            image 15 40 1 59 184
            repeatS TRUE
            repeatT TRUE
        }
    }
    geometry Box {
        size 1.5 2.2 0.5
    }
}
PixelTexture { image 2 1 1 0 255}
```

Se especifica una textura con 15 pixels de ancho y 40 pixel de alto. El primer pixel es negro y el segundo pixel es blanco.

3.16.5. Nodo TextureTransform

Este nodo define una transformación 2D aplicada a las coordenadas de textura. Esto afecta a la forma en que se aplica la textura a las superficies de los objetos. La transformación consiste (por orden) en un ajuste de la escala no uniforme sobre un punto central arbitrario, una rotación sobre ese mismo punto y una translación. Esto permite al usuario modificar el tamaño, orientación y posición de las texturas de los objetos.

```
TextureTransform{
    center Eje_S Eje_T
    rotation ángulo
    scale Eje_S Eje_T
    translation Eje_S Eje_T
}
```

center	Especifica un punto en el sistema de coordenadas de la textura (S,T) sobre el que los campos rotation y scale van a ser aplicados.
scale	Contiene dos factores de escala, uno para el eje S y otro para el eje T de la textura. Puede tomar cualquier valor real en ambos ejes.
rotation	Determina la rotación en radianes de los ejes de coordenadas de la textura con respecto al punto marcado por el campo center, después de haber aplicado el campo *scale*.
translation	Provoca un desplazamiento del sistema de coordenadas de la textura.

Es importante destacar que todas estas modificaciones aparecen invertidas cuando la textura se aplica sobre la superficie de un objeto.

```
#VRML V2.0 utf8
# Ejemplo de caja con textura de una imagen
Shape {
    appearance Appearance {
        texture ImageTexture {
            url ["eisic.jpg "]
        }
        textureTransform TextureTransform {
            scale 2 3
            rotation 0.5
            center 1 1
            translation 3 2
        }
    }
    geometry Box {
        size 1.5 2.2 0.5
    }
}
```

3.16.6. Ejercicio práctico

Se propone como ejercicio práctico poner textura a algunos de los objetos del ejercicio del capítulo anterior (Sección 13.15.3), de la siguiente manera:

Planeta VRML, adjudique a este objeto como textura (bosque.jpg). Planeta WebMaestro: adjudique a este objeto como textura la imagen (mar.jpg). Planeta Web3D: coloque a este objeto como textura la imagen (bosque.jpg).

Para ello, tal como se ha visto anteriormente, basta con sustituir, en el nodo Shape de ambos objetos el campo material:

```
material Material {}
```

por el campo texture, de la siguiente manera:

```
texture ImageTexture {
    url "imagen.gif"
}
```

Una vez realizados los cambios el código en el programa Progra27.wrl queda de la siguiente forma:

```
#VRML V2.0 utf8
#Progra27.wrl
#Aplicación del nodo Transform
Shape {
    #Campo appearance:
    appearance Appearance {
        material Material {
            diffuseColor 1 0.5 1
            transparency 0.5
        }
    }
    #Campo geometry:
    geometry Sphere {
        radius 10
    }
}
Transform {
    translation 10.0 6.0 10.0
    children [
        Shape{
            appearance Appearance {
                material Material {
                    diffuseColor 1 0.3 0.3
                    transparency 0.6
                }
            }
            geometry Sphere {
```

```
                radius 1
            }
        },
    ]
}
Transform {
    translation 25.0 1.0 -7.0
    children [
        Shape{
            appearance DEF Fondo Appearance {
                material Material{}
            }
            geometry Sphere {
                radius 1
            }
        },
    ]
}
Transform {
    translation -0.8 8.0 27.0
    children [
        Shape{
            appearance Appearance {
                texture ImageTexture {
                    url "mar.jpg"
                }
            }
            geometry Box {
                size 1 1 1
            }
        },
    ]
}
Transform {
    translation -17.0 14.0 25.0
    children [
        Shape{
            appearance Appearance {
                material Material {
                    diffuseColor 1 1 0
                }
            }
            geometry Text {
                string ["BIENVENIDOS",
                "A NUESTRA GALAXIA VIRTUAL"]
                fontStyle FontStyle {
                    family "ARIAL",
                    style "BOTH",
                    size 5.0
                    spacing 1.0
                    justify "MIDDLE"
                }
            }
        },
    ]
}
Transform {
    translation 7 1 12.0
    children [
        Shape{
            appearance USE Fondo
```

Figura 3.28. Colocación de imágenes en los Planetas

```
            geometry Text {
                string ["VRML"]
                fontStyle FontStyle {
                    family "ARIAL",
                    style "BOTH",
                    size 4.0
                    spacing 1.0
                    justify "MIDDLE"
                }
            }
        },
    ]
}
Transform {
    translation -1.8 -8.0 20.0
    children [
        Shape{
            appearance Appearance {
                texture ImageTexture {
                    url "bosque.jpg"
                }
            }
            geometry Cone {
                height 1.5
                bottomRadius 1.0
            }
        },
    ]
}
```

Como todavía no se ha definido el punto de vista inicial más conveniente, puede ser que el visor se sitúe inicialmente dentro de la esfera mayor, y no se vea nada. En este caso, es necesario retroceder con los controles del visor.

3.17. ILUMINANDO EL ESCENARIO VIRTUAL

Hasta ahora sé ha visto cómo crear objetos, darles colores y texturas, modificarlos y combinarlos para crear escenarios más o menos complejos. Ahora se necesita luz para verlos.

VRML facilita varios nodos para definir luces. Sin embargo, las luces en VRML no son iguales a las del mundo real. En el mundo real, siempre existe un objeto que emite luz, y se puede ver el objeto y la luz que emite.

Por el contrario, las luces en VRML no tienen un objeto visible asociado. Por lo tanto, tener una fuente de luz muy cerca de un objeto, sin que se vea para nada la fuente de esa luz, solamente sus efectos sobre el objeto. Otra diferencia con el mundo real, es que los objetos iluminados no proyectan sombras. Esto es debido a que la luz no se trata como un flujo de fotones. A cada plano de cada objeto se le aplicará una ecuación de luz que tendrá en cuenta todas las luces de la escena y calcula el color que finalmente presentará ese plano.

Por defecto, las escenas de los escenarios se iluminan con un foco situado en la cabeza del navegante (headlight). De todos modos, VRML permite definir tres tipos adicionales de luz:

- Luz direccional.- Se trata de una iluminación con rayos paralelos que vienen del infinito. Por lo tanto, no se indica la posición de la fuente de luz, sino simplemente su dirección.
- Luz puntual.- Se trata de una iluminación mediante un punto de luz. Los rayos parten del punto de luz en todas direcciones. Para este tipo de iluminación se especifica, entre otros parámetros, la posición de dicho punto de luz.
- Foco.- Este tipo de luz funciona como una linterna o faro de un auto. La iluminación es mayor en determinada dirección, y se va atenuando según se va alejando de dicha dirección.

Ahora se verá con más profundidad cada uno de los tipos de nodos de luz disponibles.

3.17.1. Nodo PointLight

Este tipo de luz, está localizado en un punto concreto del espacio, e ilumina en todas direcciones. La sintaxis de este nodo es:

```
PointLight{
    color color_RGB
    location Eje_X Eje_Y Eje_Z
    radius valor_real
    attenuation coeficiente1 coeficiente2 coeficiente3
    on valor_lógico
    intensity valor_real
    ambientIntensity valor_real
}
```

Los campos específicos de este tipo de luz son:

color	Indica el color de la luz, que se mezclará con el color de los objetos sobre los que se colocara la luz.
location	Esta es la posición de la luz en el sistema de coordenadas local.
radius	Especifica el radio de acción de la luz. Los objetos que se encuentren a una distancia superior de la fuente de luz, no serán afectados por la misma.
attenuation	Indica el grado de atenuación. Se compone de tres valores. El primero indica el grado de atenuación constante, el segundo la atenuación lineal con la distancia y el tercero la atenuación cuadrática. Este último tipo es el más realista, pero también el más difícil de calcular por el navegador. El factor de atenuación viene dado por la siguiente ecuación: 1/(coef1+(coef2*radio)+ (coef3*radio*radio))
on	Muestra si los campos intensity y ambientIntensity están activos (TRUE) o no (FALSE).

intensity	Establece la intensidad con la que la luz brilla, siendo el valor 1 la máxima intensidad y 0 la mínima.
ambientIntensity	Determina la luminosidad del entorno del foco.

Un ejemplo para observar el funcionamiento de este nodo es el siguiente:

```
#VRML V2.0 utf8
PointLight {
    location 8 -3 -10
    radius 15
    ambientIntensity 0.5
    color 0 0 1
    intensity 1
    attenuation 1 0.5 0 # Presenta una cierta atenuación
}

Transform {
    translation 0 -5 -10
    children Shape {
        appearance Appearance {
            material Material {
                ambientIntensity 1
                diffuseColor 1 1 1
            }
        }
        geometry Box {
            size 20 1 10
        }
    }
}
```

3.17.2. Nodo DirectonialLight

Define una fuente de luz orientable que ilumina con rayos paralelos a un determinado vector tridimensional. Siempre se debe indicar el sentido hacia donde van los rayos, y no desde donde vienen.

Este tipo de luces no exige mucha potencia de procesado al ordenador, y por lo tanto son las más recomendables para iluminaciones generales.

```
DirectionalLight{
    color color_RGB
    on valor_lógico
    intensity valor_real
    ambientIntensity valor_real
    direction Eje_X Eje_Y Eje_Z
}
```

direction	Contiene el vector que determina la orientación de la luz emitida dentro del espacio virtual. El resto de campos tienen la misma misión que en el nodo PointLight.

Aquí tiene un ejemplo práctico:

```
#VRML V2.0 utf8
# Ejemplo de iluminación con tres colores
DirectionalLight { # rojo
    ambientIntensity 0.5
    color 1 0 0
    intensity 1
    direction 0.5 -0.5 -0.1
}
DirectionalLight { # verde
    ambientIntensity 0.5
    color 0 1 0
    intensity 1
    direction -0.5 -0.5 -0.1
}
DirectionalLight { # azul
    ambientIntensity 0.5
    color 0 0 1
    intensity 1
    direction 0 0.5 -0.1
}
Shape {
    appearance Appearance {
        material Material {
            ambientIntensity 1
            diffuseColor 1 1 1
        }
    }
    geometry Sphere {
        radius 2
    }
}
```

3.17.3. Nodo SpotLight

Define una fuente de luz de tipo foco, que se coloca en una posición fija del espacio tridimensional e ilumina en forma de cono a lo largo de una dirección determinada. La intensidad de la iluminación desciende de forma exponencial según diverge el rayo de luz desde esa dirección hacia los bordes del cono. El régimen de descenso y el ángulo del cono se controlan mediante los campos beamWidth y cutOffAngle.

```
SpotLight{
    color color_RGB
    location Eje_X Eje_Y Eje_Z
    radius valor_real
    attenuation coeficiente1 coeficiente2 coeficiente3
    on valor_lógico
    intensity valor_real
    ambientIntensity valor_real
    direction Eje_X Eje_Y Eje_Z
    beamWidth ángulo
    cutOffAngle ángulo
}
```

beamWidth	Almacena el radio (en radianes) de la base de un cono donde la luz emitida es uniforme y posee su máxima intensidad. Este cono tendría como base este campo, como altura el campo radius (orientado según el campo direction) y como vértice el punto indicado en el campo location.
cutOffAngle	Almacena el radio (en radianes) de la base de un cono que contiene al cono mencionado arriba. Tiene las mismas características a excepción de su radio, el cual ha de ser mayor. Este radio determina el lugar donde la luminosidad es nula. Entre el radio almacenado en beamWidth y el almacenado en este campo, la intensidad de la luz va decreciendo conforme se aleja del primero de los conos.

Un ejemplo para el uso de este nodo se presenta a continuación:

```
#VRML V2.0 utf8
SpotLight {
    location 0 0 -10
    radius 3
    ambientIntensity 0.5
    color 1 0 0
    intensity 1
    attenuation 1 0 0
    direction 0 -1 0
    cutOffAngle 1.0472 # 60 grados en radianes
    beamWidth 0.785398 # 45 grados en radianes
}

Transform {
    translation -10 -4 -15
    children Shape {
        appearance Appearance {
            material Material {
                ambientIntensity 1
                diffuseColor 1 1 1
            }
        }
        geometry ElevationGrid {
            colorPerVertex FALSE
            xDimension 20
            zDimension 10
            xSpacing 1
            zSpacing 1
            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 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
                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            ]
        }
    }
}
```

3.18. REUTILIZACION Y PROTOTIPOS

En esta sección se presenta tres facilidades que proporciona VRML para poder hacer un uso más eficiente de los objetos que se definen para los escenarios virtuales. Estos dos elementos, junto con la posibilidad de describir estructuras jerárquicas, van a permitir definir escenas organizadas lógicamente de manera sencilla.

3.18.1. Reutilización

Cuando se necesitan varios objetos exactamente iguales, no es necesario duplicar los nodos que los definen. Existe una forma de nombrar los nodos para luego referenciarlos usando su nombre. Para hacer esto, solamente hay que usar la palabra DEF, seguida del nombre elegido para el nodo, antes de la implementación del nodo.

Para usar el mismo nodo otra vez, solamente hay que escribir la palabra USE, seguida del nombre del nodo, en el lugar donde aparecería el nodo directamente. Esto se llama, crear otra instancia del nodo.

Por supuesto, el tipo del nodo instanciado deberá ser compatible con el campo en el que se use, es decir, no se puede usar un nodo *Material* en un campo geometry.

Hay que tener en cuenta que, si se modifica el nodo original (el que va precedido por DEF), todas sus instancias se verán también modificadas.

El uso de DEF y USE es muy recomendable, no solo porque facilita la implementación y lectura de los ficheros, sino porque los hace más cortos (y más rápidos de descargar de la Web) y permiten que el navegador ahorre memoria. Especialmente recomendable es su uso con texturas de todo tipo, ya que éstas suelen ser ficheros grandes.

3.18.2. Prototipos

Anteriormente sé ha visto cómo reutilizar objetos exactamente iguales. Si se deseara implementar objetos parecidos pero con algunas diferencias, tendría que implementarlos por separado. Este problema se puede resolver mediante los prototipos.

En principio, los prototipos puede parecer una simple variación del concepto de reutilización (DEF/USE). Sin embargo son cosas muy diferentes. Cuando se crea un prototipo se está creando un nuevo tipo de nodo, no un nodo. Para crear un nodo del nuevo tipo, (una instancia de ese nodo) se realiza lo mismo que con los nodos estándar.

Para definir un prototipo, se usa la declaración PROTO y se siguen los siguientes pasos:

- Primero se da un nombre al nuevo tipo.
- Luego se enumeran sus campos y eventos entre corchetes (se explicará los eventos más adelante), indicando el tipo de cada uno y su valor por defecto.
- Finalmente, se define el nuevo tipo como combinación de otros tipos ya conocidos. Estos deberán ir entre llaves.

A continuación se presenta un ejemplo de la declaración de un prototipo.

```
#VRML V2.0 utf8
#Ejemplo, utilización de PROTO
PROTO Columna[
    exposedField SFColor colorColumna .5 .5 .5
]
{
    Shape{
        appearance Appearance{
            material Material{
                diffuseColor IS colorColumna
            }
        }
        geometry Cylinder{}
    }
}
Shape{
    appearance Appearance{
        material Material{
            diffuseColor 1 0 1
        }
    }
    geometry Box{}
}
Columna{
    colorColumna 1 0 0
}
```

La línea diffuseColor IS colorColumna indica que el valor que se le da al campo colorColumna en cada instancia del prototipo será aplicado al campo diffuseColor del nodo Material. Por ejemplo, si escribe en el fichero

```
Columna { colorColumna 1 1 0 }
```

Tendría una columna amarilla.

El tipo de cada instancia del prototipo será el mismo que el primer nodo que aparezca en el campo de la implementación (entre llaves). Por tanto, dichas instancias solamente podrán aparecer en el fichero en los sitios en que está permitido que aparezcan los nodos de los que

hereda el tipo. En el ejemplo anterior, las instancias del prototipo Columna solamente podrán estar en los sitios donde antes podía aparecer un nodo Shape.

Es necesario comentar que no se puede acceder a nodos que estén dentro del prototipo. Por ejemplo, si un nodo está nombrado con DEF fuera del prototipo, no se puede usar dentro, o viceversa. Solamente la interfaz pública (los campos y eventos) permite el contacto entre el interior y el exterior del prototipo.

Se puede usar un prototipo que haya sido declarado en otro fichero. En ese caso, se tiene que declarar la interfaz pública también, pero no los valores por defecto de los campos. La referencia al otro fichero se hace mediante la palabra EXTERNPROTO, indicando en qué fichero se encuentra la definición del prototipo, y cual de los prototipos del fichero es el que se busca (sí hay varios). Por ejemplo:

```
EXTERNPROTO Columna [
    field SFColor colorColumna
]
"columna.wrl#Columna"
```

3.18.3. Nodo Inline

Este nodo busca sus nodos hijos en cualquier lugar de la Web. De esta forma, el escenario virtual puede ser dividido en archivos, de forma que se puedan desarrollar por separado. Asimismo, esto permite usar ficheros implementados por otras personas o almacenados en el propio sistema local. Su sintaxis es:

```
Inline{
    url [ ... ]
    bboxCenter Eje_X Eje_Y Eje_Z
    bboxSize -1 -1 -1
}
```

El campo url contiene el URL del fichero que contiene al hijo o hijos del nodo Inline. Se pueden especificar varios URL'S en orden de preferencia. Algunos navegadores, no buscan y cargan ese archivo hasta que el nodo Inline se hace visible.

3.19. NIVELES DE DETALLE

Definir objetos con diferentes niveles de detalle va a permitir una mayor optimización del escenario virtual, ya que los objetos más lejanos al visitante se representan mediante formas más simples de las que tendrían si se estuviese junto a ellos. Otra de sus ventajas es que se reduce el tiempo de carga del escenario VRML.

3.19.1. Nodo LOD (Level Of Detail)

Este nodo se utiliza para permitir a los navegadores conmutar automáticamente entre varias presentaciones de objetos. Los hijos de este nodo representan generalmente el mismo objeto u objetos, a distintos niveles de detalle, que van variando desde el superior al inferior. Su sintaxis es:

```
LOD{
    center Eje_X Eje_Y Eje_Z
    range [valor1,valor2,...,valorN]
    level [Nodo1,Nodo2,...,NodoN,NodoN+1]
}
```

center	Determina la posición que va a ocupar el objeto LOD dentro del sistema de coordenadas.
level	Contiene una lista de nodos que representan a un mismo objeto con diferentes niveles de detalle, definiéndose las representaciones de mayor nivel al principio de la lista y los de menor nivel de detalle al final.
range	Contiene una lista de valores en orden creciente que indican a qué distancia se ha de conmutar entre una representación u otra.

Para calcular la conmutación, primero se calcula la distancia que hay entre el visitante y el punto central especificado del LOD; si es menor que el primer valor de la lista range entonces se dibuja el primer hijo de LOD indicado en la lista level; si está entre el primer y el segundo valor de la lista range, se dibuja el segundo hijo, y así sucesivamente. Si en la lista range figura N valores, la lista level ha de tener N+1 hijos.

Cada valor del campo range debe ser menor que su predecesor, ya que de no ser así, los resultados serían indefinidos.

No se deben usar estos nodos para emular comportamientos, ya que los resultados pueden no coincidir con el efecto deseado. Por ejemplo, la utilización de un LOD para hacer que una puerta se abra cuando se aproxima un usuario es posible que no dé resultado en todos los navegadores.

3.20. EVENTOS Y ANIMACIONES

Una de las mejoras, que VRML 2.0 presenta frente a la versión anterior es la capacidad para añadir un comportamiento a los objetos. Estos pueden moverse, responder a órdenes, etc. Esto se consigue mediante las animaciones y la interacción con el usuario.

Pero antes de empezar a describir las animaciones en VRML, es preciso que se explique un concepto básico que es:

3.20.1. Eventos

Según lo visto hasta ahora, en los campos de los objetos se ponían valores que definían sus características. Pero esos valores no podían cambiar con el tiempo: siempre contenían los valores que el diseñador les daba al escribir el código VRML.

Un evento es una petición (una orden, una señal) para que un determinado campo de un objeto cambie su valor. Los eventos pueden ser de entrada (*EventIn*), que cambian el valor del campo al que entran, o de salida (*EventOut*), que envían un valor a algún campo de otro nodo.

Para que esos campos puedan ser cambiados mediante eventos, deberán ser definidos con el tipo *exposedField*. Esto define automáticamente dos eventos: un evento de entrada llamado *set_nombreDelCampo,* que modificará el valor de ese campo cuando se produzca dicho evento, y otro de salida, llamado *nombreDelCampo_changed*, y que enviará el nuevo valor cuando el campo al que está asociado cambia de valor.

Muchos campos de los tipos de objetos definidos por VRML tienen asociados eventos set_ y _changed, como el campo rotation del nodo Transform, por ejemplo. Se llama a estos campos "campos expuestos" (*exposed fields*). Algunos campos en ciertos nodos sólo tienen asociado un evento de entrada.

Se pueden definir eventos de entrada o salida que actúen sobre campos que no están en la interfaz pública del prototipo o eventos que no estén asociados a ningún campo, pero esto sólo es posible mediante el uso de scripts.

3.20.2. Animación e interacción con el usuario

Las animaciones, permiten que los objetos cambien sus características (sus campos) con el tiempo. Hay que destacar aquí que dichos cambios se producen en la representación interna del escenario virtual en el navegador en tiempo de ejecución, no en el fichero VRML que describe ese escenario. Los cambios se producen mediante eventos que han sido enrutados hacia los campos. El fichero VRML define qué eventos se producirán y a qué campos afectarán, pero el control de los eventos siempre lo lleva el navegador.

Los eventos se enrutarán, entre sí, de forma que un evento de salida le pase su valor a un evento de entrada. Esto se hará mediante las palabras reservadas ROUTE y TO de la siguiente forma:

ROUTE Nodo1.evento_salida TO Nodo2.evento_entrada

Nodo1 y Nodo2 han de ser nombres dados a los nodos mediante la palabra DEF, esto implica que todas las instancias de ese nodo (USE) cambiarán también el valor de los campos. Esto se debe a que, en realidad, todas las instancias de un objeto son el mismo objeto repetido. Si cambia el original, cambian las copias.

Los tipos de los eventos enrutados de esta forma han de ser iguales. Los eventos usan los mismos tipos que los campos (booleano, real, etc.).

Por ejemplo ROUTE puede modificar el color de una iluminación direccional en función del cambio de color de un material.

```
#VRML V2.0 utf8
#Ejemplo del uso del nodo ROUTE
Shape {
    geometry Sphere {
        radius 2
    }
    appearance Appearance {
        material DEF MaterialEsfera Material {
            diffuseColor 0 0 1
        }
    }
}
DEF Iluminacion DirectionalLight {
    color 1 1 1
    direction 0.5 0.5 0
    intensity 1
}
ROUTE MaterialEsfera.diffuseColor TO Iluminacion.color
```

En el ejemplo se ha aplicado la iluminación a la esfera, en este caso cualquiera que sea el color de la misma la iluminación va ha ser la misma, esto se puede aplicar a cualquier objeto utilizando el comando ROUTE.

3.20.3. Disparadores

El disparador suele ser un nodo sensor, que permiten detectar la proximidad del usuario a un área determinada.

Estos sensores, detectan las acciones del usuario sobre objetos determinados, que han de estar dentro del mismo grupo (cualquier tipo de nodo de agrupamiento: Group, Transform, etc.) que el detector, es decir han de ser hijos del mismo nodo agrupador. Si no se hace así, el detector no funcionará en absoluto, es decir, para animar un objeto el procedimiento habitual comienza por definir el objeto y su disparador como hijos de un nodo de agrupamiento. Sin embargo, los

nodos a los que van enrutados los eventos generados por el sensor pueden estar en cualquier parte del fichero.

Los disparadores son invisibles. Simplemente detectan cambios en una geometría sencilla, y generan los eventos correspondientes. A continuación los nodos sensores disponibles en VRML son:

3.20.3.1. Sensor de Visibilidad VisibilitySensor

El nodo VisiblitySensor detecta cuándo el usuario puede ver un objeto o área determinados. Su sintaxis se puede observar a continuación:

```
VisibilitySensor{
    center Eje_X Eje_Y Eje_Z
    size 0 0 0
    enabled Valor_lógico
}
```

center	Es el centro del área que activa y desactiva el sensor.
size	Este es el tamaño en las tres dimensiones del área.
enabled	Indica, como en ejemplos anteriores, si el sensor está activo.

Los eventos de salida de este sensor son los siguientes:

isActive	Muestra si la región que interesa está a la vista (TRUE) o no (FALSE).
enterTime	Es el tiempo en el que, es generado el evento isActive con valor TRUE.
exitTime	Es el tiempo en el que, es generado el evento isActive con valor FALSE.

Para una mejor compresión se expone un ejemplo:

```
#VRML V2.0 utf8
#Ejemplo1 del Nodo VisibilitySensor
Transform {
    translation -5 8 3
    children Shape {
        geometry Sphere {
            radius 2
        }
        appearance Appearance {
            material Material {
                diffuseColor 0.7 0.3 0
            }
        }
    }
}
```

```
VisibilitySensor {
    center -5 8 3
    size 4 4 4
}
```

3.20.3.2. Sensor de Movimiento CylinderSensor

Este nodo permite que el usuario gire mediante arrastres de ratón. El efecto es el mismo que si el usuario girara un cilindro sobre su eje (de ahí el nombre del sensor). A continuación se ve su sintaxis:

```
CylinderSensor{
    minAngle ángulo
    maxAngle ángulo
    enabled Valor_lógico
    diskAngle ángulo
    offset ángulo
    autoOffset Valor_lógico
}
```

minAngle	Es el ángulo más pequeño que puede entregar a un evento rotation_changed. Cualquier ángulo menor será convertido en minAngle.
maxAngle	Es el equivalente pero para el ángulo mayor. Ambos campos limitan el rango en el que se puede producir la rotación del objeto.
enabled	Indica si el sensor está prestando atención a los eventos producidos por el dispositivo usado (ratón en general). Se puede cambiar con el evento set_enabled.
diskAngle	Determina si el sensor se comporta como un cilindro o como un disco. Si el usuario pincha el ratón cerca del eje del cilindro, el sensor se comporta como un disco, caso contrario se comporta como un cilindro.
offset	Indica en cuántos radianes girará el objeto asociado desde su orientación inicial, cada vez que el usuario comienza a arrastrar el sensor de nuevo.
autoOffset	Indica si el navegador debe tomar nota de la orientación final en cada operación de giro, salvándola en el campo offset. Si la orientación no se guarda, el objeto vuelve a su posición inicial al comenzar un nuevo arrastre.

El cilindro imaginario que el navegador genera para manejar los giros tiene su eje en el eje Y del sistema de coordenadas local. Realmente, lo que da potencia a un nodo usado en una animación son los eventos que puede manejar. Este nodo en concreto puede manejar tres:

isActive	Es un evento de salida que indica si en el momento actual el usuario esta arrastrando la geometría.
trackPoint_changed	Otro evento de salida que indica el punto del cilindro imaginario al que el usuario está apuntando en un momento dado.
rotation_changed	También es un evento de salida, que toma como valor la orientación actual del cilindro.

Un ejemplo práctico de su utilización es el siguiente:

```
#VRML V2.0 utf8
#Ejemplo del Nodo CylinderSensor
Transform { # Cub de referencia
    translation -3 0 0
    children Shape {
        geometry Box {
            size 2 2 2
        }
        appearance Appearance {
            material Material {
                diffuseColor 0.7 0.3 0
            }
        }
    }
}
Group {
    children [
        DEF Cub Transform {
            children Shape {
                geometry Box {
                    size 2 2 2
                }
                appearance Appearance {
                    material Material {
                        diffuseColor 0 0.7 0.3
                    }
                }
            }
        }
        DEF CS CylinderSensor { }
    ]
}
ROUTE CS.rotation_changed TO Cub.rotation
```

Si ejecuta este ejemplo podrá observar dos objetos, en donde uno de los objetos se mueve al momento de hacer un clic con el Mouse.

3.20.3.3. Sensor de Movimiento PlaneSensor

Este nodo interpreta la información dada por el dispositivo (normalmente el ratón) como desplazamientos sobre el plano XY del sistema de coordenadas local. Su sintaxis es:

```
PlaneSensor{
    minPosition Eje_X Eje_Y
    maxPosition Eje_X Eje_Y
    enabled Valor_lógico
    offset 0 0 0
    autoOffset
}
```

minPosition	Indica un punto en el plano XY que es la esquina inferior izquierda del rectángulo en el que se podrá mover el objeto animado.
maxPosition	Muestra el punto límite superior derecho del citado rectángulo.
enabled	Es un campo booleano que determina si el sensor está prestando atención a los eventos producidos por el dispositivo usado por el usuario.
offset	Contiene una traslación en el plano XY desde la posición inicial que se aplicará cada vez que se empiece un nuevo arrastre del objeto.
autoOffset	Es equivalente al campo homónimo del nodo CylinderSensor. Indica si conserva la posición final de la translación anterior al comenzar una nueva.

Y sus eventos:

isActive	Indica si el ratón tiene oprimido su botón en el momento actual. Este evento solamente se genera al pulsar o soltar el botón, no durante el arrastre del mismo.
trackPoint_changed	Indica el punto del plano XY al que el ratón está apuntando. Los valores minPosition y maxPosition son ignorados.
translation_changed	Es el punto del plano XY al que apunta el ratón durante el arrastre.

A continuación un ejemplo práctico del uso del Nodo Sensor, en el cual se observara dos cubos un verde y un naranja, al naranja es al que se le ha aplicado este sensor, el mismo que tendrá la facultad de moverse alrededor del cubo verde, el código es el siguiente:

```
#VRML V2.0 utf8
#Ejemplo del Nodo PlaneSensor
Transform { # Cubo de referencia
    translation -3 0 0
    children Shape {
        geometry Box {
            size 2 2 2
        }
        appearance Appearance {
            material Material {
                diffuseColor 0.7 0.3 0
            }
        }
    }
}
Group {
    children [
        DEF Cub Transform {
            children Shape {
                geometry Box {
                    size 2 2 2
                }
                appearance Appearance {
                    material Material {
                        diffuseColor 0 0.7 0.3
                    }
                }
            }
```

```
        }
        DEF PS PlaneSensor { }
    ]
}
ROUTE PS.translation_changed TO Cub.translation
```

Este es un ejemplo práctico y muy eficiente.

3.20.3.4. Sensor de Movimiento SphereSensor

Este sensor también produce giros, pero esta vez el objeto se moverá alrededor de un punto, y no alrededor de un eje, a continuación su sintaxis es:

```
SphereSensor{
    enabled Valor_lógico
    offset 0 1 0 0
    autoOffset Valor_lógico
}
```

enabled	Indica si el sensor presta atención al dispositivo de entrada.
offset	Contiene el valor de la rotación que se le dará al objeto desde la posición inicial al producirse un nuevo arrastre del usuario.
autoOffset	Indica si la posición del objeto se mantendrá entre distintas intervenciones del usuario.

Los eventos de salida de este nodo (isActive, trackPoint_changed y rotation_changed) tienen exactamente la misma utilidad que en el caso del nodo CylinderSensor, por lo tanto no se volverán a describir.

El ejemplo Progra28.wrl muestra como utilizar esto nodo y además el manejo del evento rotation_chaged.

```
#VRML V2.0 utf8
#Progra28.wrl
#Ejemplo1 del Nodo SphereSensor
Transform { # Cubo de referencia
    translation -3 0 0
    children Shape {
        geometry Box {
            size 2 2 2
        }
        appearance Appearance {
            material Material {
                diffuseColor 0.7 0.3 0
            }
        }
    }
}
Group {
```

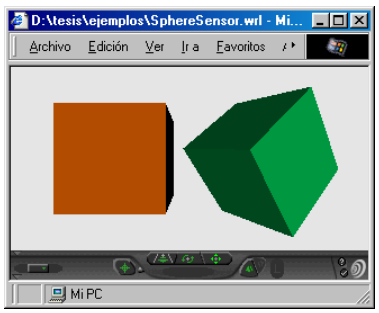


Figura 3.29. Simulación de animación

```
    children [
        DEF Cub Transform {
            children Shape {
                geometry Box {
                    size 2 2 2
                }
                appearance Appearance {
                    material Material {
                        diffuseColor 0 0.7 0.3
                    }
                }
            }
        }
        DEF SS SphereSensor { }
    ]
}
ROUTE SS.rotation_changed TO Cub.rotation
```

Como se ve en la gráfica, el cubo verde es el que va realizar el evento rotar en su mismo eje, si el usuario lo requiere, al hacer uso del ratón.

3.20.3.5. ProximitySensor

Como su nombre indica, este sensor detecta los movimientos del usuario dentro de un área, permitiendo detectar las entradas y salidas de esa área y ciertos movimientos dentro de ella. Su sintaxis es:

```
ProximitySensor{
    center Eje_X Eje_Y Eje_X
    size 0 0 0
    enabled Valor_lógico
}
```

center	Es el centro de la región del espacio en la que el sensor detectará los movimientos del usuario.
size	Define la extensión de la región alrededor del centro en las direcciones de los ejes.
enabled	Como en los casos anteriores indica si el sensor está activo.

En este caso el número de eventos son cinco, todos ellos de salida:

isActive	Indica si el usuario ha entrado en la región (TRUE) o si ha salido (FALSE).
position_changed	Es la posición actual del usuario, que se actualiza cada vez que este entra o se mueve dentro de la región.
orientation_changed	Es la orientación actual del usuario. Se actualiza igual que en el caso anterior.
enterTime	Es el momento exacto que el usuario entra en la región.

exitTime	Es el momento exacto que el usuario sale de la región.

En el siguiente ejemplo se observa el funcionamiento de este Sensor:

```
#VRML V2.0 utf8
#Ejemplo del Nodo ProximitySensor
DEF motorColor TimeSensor{
    loop TRUE
    cycleInterval 5
}
DEF ColoresCubo ColorInterpolator {
    key      [ 0,   0.3,   0.6,     1]
    keyValue [ 1 0 0, 0 1 0, 0 0 1, 1 0 0]
}
DEF CambiaColorCubo Shape {
    geometry Box {
        size 2 2 2
    }
    appearance Appearance {
        material DEF materialCambiaColor Material {}
    }
}
DEF ZonaActiva ProximitySensor {
    center 0 0 0
    size 2 2 2
}
ROUTE motorColor.fraction_changed TO ColoresCubo.set_fraction
ROUTE ColoresCubo.value_changed TO materialCambiaColor.diffuseColor
ROUTE ZonaActiva.isActive TO motorColor.enabled
```

Como se puede observar se utiliza un cubo el mismo que cambia de color al detectar la entrada del usuario al escenario virtual.

3.20.3.6. TouchSensor

Este sensor indica si el usuario está apuntando o accionando un objeto. Sólo tiene un campo, para activarlo o desactivarlo. Toda su funcionalidad se consiguen haciendo uso de sus eventos.

```
TouchSensor {
    enabled Valor_lógico
}
```

Este sensor tiene seis eventos de salida:

isOver	Indica si el ratón está apuntado al objeto asociado al sensor, independientemente de que el botón esté presionado o no.
isActive	Indica si el botón está actualmente presionado.

hitPoint_changed	Es la localización en la superficie de los objetos asociados al sensor en el que el botón fue soltado por última vez.
hitNormal_changed	Es lo normal en el punto dado por hitPoint_changed.
hitTexCoord_changed	Es el punto de coordenadas de la textura en el punto dado por hitPoint_changed.
TouchTime	Da el momento en el que se completa el clic del ratón.

Ejemplo:

```
#VRML V2.0 utf8
#Progra29.wrl
#Ejemplo del Nodo ProximitySensor
DEF motorColor TimeSensor{
    loop TRUE
    cycleInterval 5
    enabled FALSE
}
DEF ColoresCubo ColorInterpolator {
    key      [     0,   0.3,   0.6,     1]
    keyValue [ 1 0 0, 0 1 0, 0 0 1, 1 0 0]
}
DEF CambiaColorCubo Shape {
    geometry Box {
        size 2 2 2
    }
    appearance Appearance {
        material DEF materialCambiaColor Material {}
    }
}
DEF BotonRojo Transform {
    translation 2 -1 4
    children [
        Shape {
            geometry Cylinder {
                height 0.1
                radius 1
            }
            appearance Appearance {
                material Material {
                    diffuseColor 1 0 0
                }
            }
        }
        DEF SensorBotonRojo TouchSensor { }
    ]
}
ROUTE motorColor.fraction_changed TO ColoresCubo.set_fraction
ROUTE ColoresCubo.value_changed TO materialCambiaColor.diffuseColor
ROUTE SensorBotonRojo.isActive TO motorColor.enabled
```

Figura 3.30. Simulación de animación con el evento TouchSensor

Como se puede observar en el gráfico se tiene un cubo y un botón, el mismo que servirá para modificar los colores del cubo, al realizar un clic sobre el botón este cambiara de color.

3.20.4. Relojes

El único nodo que puede hacer las veces de reloj es el nodo TimeSensor. Su sintaxis se muestra a continuación:

```
TimeSensor{
    cycleInterval 1
    enabled Valor_lógico
    loop Valor_lógico
    startTime 0
    stopTime 0
}
```

cycleInterval	Es el tiempo de la animación en segundos. Si se repite la animación cíclicamente, será la duración de cada ciclo.
enabled	Indica si el reloj está en funcionamiento o no. Si está, se generan los eventos relacionados con el tiempo cuando sea oportuno, caso contrario no se generarán bajo ninguna circunstancia. Independientemente de esto, se procesan los eventos set_ y se generan los eventos _changed.
loop	Indica si la animación se repetirá indefinidamente (hasta llegar al momento de parada) o si solamente se parará después de un ciclo.
startTime	Momento en el que se empiezan a generar eventos.
stopTime	Momento en el que se dejan de generar eventos.

Los eventos de salida son cuatro:

isActive	Envía el valor TRUE cuando el sensor empieza a funcionar y FALSE cuando se detiene.
cycleTime	Es el tiempo o la hora actual, enviado al comienzo de cada ciclo.
fraction_changed	Envía la fracción del ciclo actual que ya ha pasado. Serán valores entre 0, al comienzo del ciclo, y 1 al final.
time	Es la hora actual, en segundos desde la medianoche del 1 de Enero de 1970.

3.20.5. Motores

La mayoría de las veces, las animaciones cambian campos que no contienen valores simples, como números reales o valores temporales. Por ejemplo, podría estar interesado en hacer girar un objeto, con lo cual el campo a cambiar sería el rotation de algún nodo Transform, que se compone de cuatro cifras. Para resolver este problema se usan los motores.

Los motores más complejos se implementan con scripts, que son pequeños programas, y que se verá más adelante. Pero para casos más sencillos puede ser suficiente un nodo interpolador.

Los nodos interpoladores definen una correspondencia entre una clave, que varía entre 0 y 1, y un valor, que depende del tipo de interpolador. Los valores no identificados en la tabla de claves y valores se interpolan. De ahí el nombre de estos nodos.

VRML tiene los siguientes interpoladores:

- ColorInterpolator: genera valores de color RGB.
- CoordinateInterpolator: genera valores de coordenadas 3D.
- NormalInterpolator: genera vectores 3D para utilizar como normales en los nodos IndexedFaceSet o ElevationGrid.
- OrientationInterpolator: genera valores de rotaciones (coordenada 3D para definir el eje de rotación, y ángulo)
- PositionInterpolator: genera coordenadas 3D que definen un camino para trasladar un nodo de un lugar a otro.
- ScalarInterpolator: genera valores en punto flotante.

Todos los nodos siguen la misma estructura:

```
XXXInterpolator {
    key []
    keyValue []
}
```

key Contiene una lista de números entre 0 y 1, que representan fracciones del ciclo de la animación (obsérvese el evento fraction_changed del nodo TimeSensor).

keyValue Contiene una lista de valores, uno por cada número en el campo key. De esta forma, el valor pasado al objeto que se va a modificar, en los momentos indicados en key, será el correspondiente del campo keyValue. El resto de los valores se interpolan.

Para estos nodos se definen dos eventos:

set_fraction Es un evento de entrada, que indica al interpolador la fracción de la animación que ya ha pasado.

value_changed Es un evento de salida que da el resultado de los cálculos del interpolador. Su valor dependerá del tipo de interpolador, y se pasará al objeto adecuado en cada momento de la animación. Tal como apuntaba antes, no se proporcionan solamente los datos en el campo key, sino todos los valores en los instantes intermedios.

3.21. UTILIZACION DE SONIDOS

La adición de sonidos al escenario le añade mayor realismo. Las principales aplicaciones del sonido son: ruidos o músicas de fondo, sonidos localizados en un determinado punto del escenario (como por ejemplo el sonido de una catarata) o sonidos que se activan cuando se realiza una determinada acción (al pulsar un interruptor, saltar una pared, etc.).

VRML utiliza dos nodos diferentes para controlar las fuentes de sonido: Sound y AudioClip. De igual manera que las luces en VRML no se comportan igual a las del mundo real, los sonidos tampoco lo hacen, como se verá a continuación.

3.21.1. Nodo Sound

Este nodo permite localizar un sonido en el espacio, y definir su radio de acción, etc. Su sintaxis es:

```
Sound{
    source AudioClip{...}
    intensity valor_real
    location Eje_X Eje_Y Eje_Z
    direction Eje_X Eje_Y Eje_Z
    inFront valor_real
    minBack valor_real
    maxFront valor_real
    maxBack valor_real
    spatialize valor_real
}
```

source	Toma como valor un nodo de tipo AudioClip, en donde se define el fichero de sonido a reproducir.
intensity	Ajusta el volumen de cada fuente de sonido. Admite únicamente valores entre 0 y 1. Una intensidad 0 es silencio y una intensidad 1 es el sonido más intenso, correspondiente al volumen del fichero original.
location	Determina en el espacio tridimensional la posición del foco de sonido.
direction	Especifica un vector tridimensional normal, en cuya dirección se emite el sonido.
minFront, minBack	Determinan, junto con la dirección, la extensión de la región (hacia delante y hacia atrás respectivamente) donde la intensidad del sonido es máxima.
maxFront, maxBack	Determinan los límites de la región a partir de la cual el sonido ya no será audible.
spatialize	Determina la forma en la que se emite el sonido. Si su valor es TRUE, el sonido parecerá provenir de un único punto. Por el contrario, si su valor es FALSE, se obtendrá un efecto envolvente, utilizado para crear sonidos ambientales

3.21.2. Nodo AudioClip

Este nodo sólamente puede aparecer en el campo source del nodo Sound. Da información sobre dónde obtener el sonido y la forma de reproducirlo. Su sintaxis se muestra a continuación:

```
AudioClip{
    url "dirección_URL"
    description "descripción_del_sonido"
    loop valor_lógico
    pitch valor_real
    startTime 0
    stopTime 0
}
```

url	Indica el URL (o URL'S) del fichero que contiene el sonido, ya sea en formato.wav o .mid.
description	Es una cadena de texto que el navegador puede presentar en lugar del sonido o además de él.
loop	Indica si el sonido se repetirá al terminar de reproducirse o no.
pitch	Es un multiplicador de frecuencia. Por ejemplo, si vale 2, el sonido se escuchará con el doble de frecuencia y el doble de rápido.
startTime	Es el tiempo en el que el sonido debería empezar a sonar.
stopTime	Es el tiempo en el que el sonido debería parar o dejar de sonar. Su valor es ignorado si es menor o igual que startTime.

Ejemplo:

```
#VRML V2.0 utf8
# Ejemplo del nodo Sound y AudioClip
Sound {
    location 0 0 0
    direction 0 0 1
    intensity 1
    minFront 1
    minBack 1
    maxFront 10
    maxBack 10
    spatialize TRUE
    source AudioClip {
        url "ambiente.wav"
        startTime 1
        loop TRUE
    }
}
Shape {
    geometry Sphere {
        radius 2
    }
    appearance Appearance {
        material Material {
```

```
            diffuseColor 1 1 0
        }
    }
}
```

3.22. CODIGO EN VRML SCRIPTS

Los scripts son programas que deben ser diseñados e implementados por el creador del escenario virtual. Al estar escritos en un lenguaje de programación general, proporcionan una gran potencia.

El nodo **Script** permite implementar formas complejas de interacción. Los usos más comunes que se le suele dar a este nodo son las siguientes:

- Trabajar como motor en una animación.
- Procesar información de entrada en el papel de lógica en una animación.
- Comunicarse con el navegador de alguna forma (por ejemplo, asociar el navegador a un punto de vista para poder animar al observador de forma que siga un camino determinado).
- Manipular la jerarquía de la escena, añadiendo o quitando nodos hijos de algún nodo agrupador.
- Comunicarse con un servidor u otro escenario VRML a través de una red.

Este nodo, como muchos otros, puede recibir eventos de entrada y generar eventos de salida, pero, a diferencia de los otros nodos, el usuario puede definir los cambios que sufrirá la información en el proceso mediante un programa. Ese programa deberá estar escrito en un lenguaje que el navegador conozca, si no, no se podrá ejecutar. Los lenguajes más usados son Java y JavaScript. La sintaxis del nodo es:

```
Script{
    url [...]
    mustEvaluate Valor_lógico
    directOuput Valor_lógico
    # Y cualquier número de:
    eventIn TipoDeEvento NombreDeEvento
    field TipoDeCampo NombreDeCampo ValorInicial
    eventOut TipoDeEvento NombreDeEvento
}
```

Los campos se explican a continuación:

url	Contiene el URL de un script a ejecutar o directamente el texto del script. Puede contener varios valores, de forma que el navegador ejecutará el primer script cuyo lenguaje conozca.

mustEvaluate	Indica si el navegador debe enviar eventos al script incluso si no se esperan salidas de este. Si su valor es FALSE, el navegador puede mejorar el rendimiento no mandando eventos al script hasta que otro nodo necesite los eventos de salida del script. Se debería dejar su valor a FALSE a menos que el script haga algo que el navegador no pueda detectar (como acceder a la red).
directOutput	Indica si el script puede cambiar la jerarquía de la escena directamente y establecer o quitar enrutados dinámicamente, o si solamente puede comunicarse con el escenario a través de eventos. Se deberá dejar a FALSE si no se necesita, para permitir que el navegador haga optimizaciones.

Se pueden incluir en el nodo tantos campos como se quiera, pero no pueden ser campos expuestos (exposedField).

No hay eventos predefinidos, pero se pueden definir los que se quieran y del tipo que se requiera de acuerdo a la necesidad de la persona que este creando el escenario virtual.

Cuando un **script** recibe un evento de entrada, le pasa el valor de dicho evento y una marca de tiempo (el momento en que ocurrió el evento de entrada) a una función o método que tiene el mismo nombre que el evento de entrada. Las funciones pueden enviar eventos de salida especificados en el nodo **Script** asignando valores a variables con el mismo nombre que los eventos. Un evento de salida enviado por un **script** tiene la misma marca temporal que el evento de entrada que lanzó la función que genera el evento de salida.

Si un **script** envía múltiples eventos con la misma marca temporal a otro nodo, el tipo del otro nodo determinará el orden en que los eventos son procesados. En general, este orden de procesado es el orden que se esperaría. Por ejemplo, si el script envía en evento set_position y un evento set_bind con la misma marca temporal a un nodo Viewpoint, el navegador cambia primero la posición y luego lo activa (bind). La mayoría de las veces las marcas temporales se ignorarán.

Algunos lenguajes (incluidos Java y JavaScript) definen los nombres de ciertas funciones o métodos que tienen un propósito especial. Si se implementa una función llamada **initialize(),** por ejemplo, se lanzará tan pronto como es cargado el escenario virtual, antes incluso de que se genere ningún evento. Si se define la función **shutdown()** se llama cuando el escenario es borrado (por ejemplo cuando el usuario activa un enlace a otro escenario virtual). La función **eventsProcessed()** se llama después de que una o más funciones relacionadas con eventos de entrada se completan; dependerá del navegador llamar a **shutdown()** después de cada función o esperar hasta que todos los eventos de entrada pendientes sean procesados.

En vez de mandar eventos a otros objetos, se puede acceder a los eventos de otros nodos definiéndolos como campos en el nodo Script. Por ejemplo en el código siguiente:

```
DEF MI_SWITCH Switch{
    whichChoice -1
}
Script{
    field SFNode nodoDirecto USE MI_SWITCH
    eventIn SFBool activate
    directOuput TRUE
    url "javascript:
    function activar(value){
        if (value==true)
        nodoDirecto.whichChoice=0;
    }"
}
```

En el ejemplo se puede ver que no necesita modificar el campo whichCoice del nodo Switch mediante un evento. Es necesario recordar que, para acceder directamente a un campo de otro nodo o para usar funciones que crean o destruyen rutas, el campo directOutput del nodo Script tiene que estar a TRUE.

La localización del nodo Script en la jerarquía de la escena no afecta a su funcionamiento. Por ejemplo, si un Script es hijo de un nodo Switch con el campo whichChoice a -1, el Script continua recibiendo y enviando eventos.

3.22.1. La interfaz del programador con el navegador

Además de interaccionar con la escena, los scripts pueden interaccionar con el navegador usando su interfaz API (Application Programer Interface), no es mas que una serie de llamadas a funciones que permiten acceder a algunos de los recursos del navegador.

A continuación se detallan los parámetros y valores de salida de estas funciones. Todos los lenguajes que permiten hacer scripts en VRML soportan estas funciones de alguna forma.

a.) Estas funciones devuelven una cadena con el nombre y versión del navegador que se está usando. Estos valores no tienen un formato definido, por lo tanto solamente se suelen usar como información. Si la información no está disponible, estas funciones devuelven una cadena vacía.

```
SFString getName();
SFString getVersion();
```

b.) Esta función devuelve la velocidad a la que el punto de vista se está moviendo en el momento actual, en metros por segundo. Si la velocidad no tiene significado en el modo actual de navegación, o si la velocidad no puede ser determinada por alguna razón, la función devuelve 0.0.

```
SFFloat getCurrentSpeed();
```

c.) Devuelve la frecuencia de los fotogramas actual, en fotogramas por segundo. Algunos navegadores no soportan esta función; si la frecuencia de fotogramas no está soportada o no puede ser ofrecida por cualquier razón, la función devuelve 0.0.

```
SFFloat getCurrentFrameRate();
```

d.) Devuelve el URL del escenario cargado en ese momento.

```
SFString getWorldURL();
```

e.) Sustituye el escenario actual con el escenario representado por los nodos pasados como parámetro. Esta función no suele retornar, puesto que el escenario anterior es perdido.

```
void replaceWorld(MFNode nodes);
```

f.) Carga el escenario dado por *url*, usando los parámetros dados. Esta función solamente retornará si el documento se carga en otra ventana u otro marco (*frame*).

```
void loadURL(MFString url, MFString parameter);
```

g.) Permite cambiar la descripción que el navegador ofrece al usuario (usualmente en la parte superior de la ventana en la que se encuentra).

```
void setDescription(SFString description);
```

h.) Recibe como parámetro una cadena que contiene la descripción de una escena VRML, analiza los nodos contenidos en la cadena y devuelve la escena correspondiente como un valor MFNode.

```
MFNode createVrmlFromString(SFString vrmlSyntax);
```

i.) Ordena al navegador que cargue un escenario VRML desde el URL dado, pero no sustituye el escenario actual con el nuevo. Después de que se carga la escena, los nodos que contiene son enviados a un evento de entrada en el nodo especificado en el campo *node*. La cadena *event* contiene el nombre del evento de entrada del nodo.

```
void createVrmlFromURL(MFString url, SFNode node, SFString event);
```

j.) Añaden y borran, respectivamente, una ruta entre los eventos dados y los nodos dados.

```
void addRoute(SFNode fromNode, SFString fromEventOut, SFNode toNode, SFString toEventIn);
void deleteRoute(SFNode fromNode, SFString fromEventOut, SFNode toNode, SFString toEventIn);
```

3.23. OTROS NODOS INTERESANTES

Algunos nodos que no se adaptan a la organización elegida en los temas anteriores son:

- Los nodos vinculables NavigationInfo, ViewPoint, Fog y Background. Estos nodos pertenecen a un grupo especial de nodos en el sentido de que solamente uno de cada tipo puede estar activo en un momento dado. En general, estos nodos proporcionan información general sobre el entorno, la escena y el usuario.
- El nodo WorldInfo. Se utiliza para dar título a un escenario virtual y para añadir a los ficheros VRML información relacionada con su documentación.

Con respecto al primer grupo de nodos (nodos vinculables, bindable nodes), siempre se activa el primero que aparece cuando se carga una escena. Los nodos activados se ponen en la cima de una pila, y hay una pila para cada tipo de nodo.

Cuando uno de estos nodos está activado, se genera un evento isBound con el valor TRUE. Para activar un nodo, se le envía mediante una ruta un evento set_Bind con el valor TRUE.

3.23.1. Nodo NavigationInfo

El nodo NavigationInfo especifica una serie de parámetros para que el navegador sepa cómo se puede mover el usuario por el escenario virtual. Su sintaxis es:

```
NavigationInfo{
    avatarSize [ 0.25, 1.6, 0.75]
    headlight Valor_lógico
    speed 1.0
    type "WALK"
    visibilityLimit 0.0
}
```

avatarSize	Da información al navegador sobre el tamaño del punto de vista activo. Esta información se necesita para el cálculo de colisiones con objetos y para seguir el contorno del suelo. El primer número indica la distancia entre la posición del observador y un objeto al que se produce la colisión. El segundo número indica a qué altura sobre el terreno se encuentra el punto de vista del observador. La tercera cifra indica la altura máxima de un objeto al que el observador se puede subir (el escalón máximo superable).
headlight	Indica si el navegador debe encender la luz frontal. Esta luz siempre apunta hacia donde está mirando el observador. Puede ser interesante en algunos casos, pero en general no es una forma muy realista de iluminación.
speed	Expresa la velocidad a la que el observador se mueve por la escena, en metros por segundo.
type	Especifica una forma de navegación. Puede tomar los valores WALK, EXAMINE, FLY o NONE.
visibilityLimit	Establece la mayor distancia a la que el observador puede ver algo. Lo que ocurra con los objetos que están a mayor distancia dependerá del navegador.

Algunos los hacen desaparecer instantáneamente, otros los van haciendo desaparecer poco a poco y otros no soportan esta opción, por lo tanto los objetos no desaparecen.

3.23.2. Nodo Viewpoint

El nodo Viewpoint define un punto concreto, en el sistema de coordenadas local, desde el que el usuario puede observar la escena. Su sintaxis es:

```
Viewpoint{
    position Eje_X  Eje_Y Eje_z
    orientation 0 0 1 0
    fieldOfView ángulo
    description "nombre del sitio"
    jump Valor_lógico
}
```

position	Especifica la localización del punto de vista en el sistema de coordenadas local.
orientation	Especifica una orientación relativa a la orientación por defecto (mirando hacia la parte negativa del eje Z).
fieldOfView	Indica el ángulo de visión del punto de vista en radianes. Cuanto mayor sea el ángulo de visión, más efecto angular se conseguirá.
description	Identifica el punto de vista para que el navegador pueda confeccionar un menú con los puntos de vista existentes. Si no se llena este campo, el navegador no añade el punto de vista al menú.
jump	Indica si el navegador cambiará de un punto de vista a otro instantáneamente (TRUE) o no (FALSE).

3.23.3. El fondo y la niebla

Los nodos Background y Fog permiten modificar el ambiente del escenario virtual, definiendo las propiedades del cielo, del suelo (y por lo tanto el horizonte) y de la atmósfera.

3.23.3.1. Nodo Background

El nodo Background permite definir el fondo del escenario virtual. Se puede indicar el color o colores del cielo y del suelo, así como seis imágenes que se colocarán como fondo. Su sintaxis es:

```
Background{
    groundColor [color_RGB]
    groundAngle [ angulo ]
    skyColor [ color_RGB ]
    skyAngle [ angulos ]
    frontUrl [ dirección_URL ]
    backUrl [ dirección_URL ]
    rightUrl [ dirección_URL ]
    leftUrl [ dirección_URL ]
    topUrl [ dirección_URL ]
    bottomUrl [ dirección_URL ]
}
```

groundColor	Contiene los valores RGB de los colores que se aplicarán al suelo, empezando por el que está justo debajo y subiendo en anillos concéntricos hasta el horizonte. El navegador hace una interpolación entre los valores dados.
groundAngle	Lista los ángulos en que se produce el cambio de un color a otro.
skyColor	Define los colores para el cielo. El primero es el color que queda justo encima del observador, y el último el del horizonte.
skyAngle	Equivalente a groundAngle pero para el cielo.
frontUrl, backUrl, rightUrl, leftUrl, topUrl, bottomUrl	Dan los URL'S de seis imágenes, que se colocarán formando un cubo de tamaño infinito.

El fondo no se ve afectado por transformaciones, excepto para la rotación. Obviamente, el usuario no podrá acercarse al fondo por más que lo intente.

El Progra30.wrl es un ejemplo de la manera de utilizar este nodo:

```
#VRML V2.0 utf8
#Progra30.wrl
#Ejemplo del Nodo Background
Background {
          #         0º       22º      45º      60º     75º        85º        90º
    skyAngle [     0       0.384,  0.785,  1.047, 1.309,    1.484,     1.5708  ]
    skyColor [  0 0 0.2,  0 0 1,  0 1 1,   0.75    0.75 1,   0.8 0.8 0,  0.8 0.6 0, 1 0.4 0 ]
    groundAngle [ 1.5708  ]
    groundColor [ 0.2 1 0.4, 0.2 1 0.4 ]
}

Shape { # Objeto de referencia
    geometry Sphere {
        radius 2
    }
    appearance Appearance {
        material Material {
            diffuseColor 1 0.8 0
        }
    }
}
```

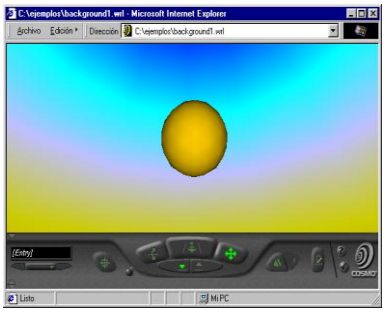


Figura 3.31. Uso del nodo Background para crear fondos.

Como muestra la gráfica se puede ver que alrededor de la esfera se observa colores diferentes de acuerdo a los ángulos que se describen arriba, esto es muy útil para realizar diferentes fondos.

Además se puede poner gráficos en vez de colores, con las funciones restantes: frontUrl, backUrl, rightUrl, leftUrl, topUrl, bottomUrl.

3.23.3.2. Nodo Fog

A continuación se puede ver la sintaxis de este nodo, que sirve para producir un efecto de niebla.

```
Fog{
    fogType "LINEAR"
    visibilityRange valor_real
    color color_RGB
}
```

fogType	Es la velocidad en que la niebla se hace más espesa al aumentar la distancia al observador. Los valores posibles son LINEAR y EXPONENTIAL.
visibilityRange	Es la máxima distancia a la que se puede ver algo a través de la niebla.
color	Es el color de la niebla.

3.23.3.3. Nodo WorldInfo

Este nodo no tiene ningún impacto visual en nuestro escenario virtual. Al igual que HTML, VRML proporciona elementos para dar un nombre a los escenarios y para añadir información que permita documentarlo. Su sintaxis es:

```
WorldInfo{
    info [ "comentario1",
    "comentario2",
    ...
    "comentarioN"]
    title "nombre_del_escenario"
}
```

title	Es una cadena que normalmente aparecerá en el borde superior de la ventana del navegador.
info	Es una lista de cadenas cuyo propósito es documentar el escenario donde está definido.

3.24. NOTAS BIBLIOGRAFICAS

Este capítulo explica lo esencial del leguaje VRML 2.0, como es su historia, sus creadores, su utilización, su estructura y su funcionamiento.

Para más información acerca de este lenguaje visite las siguientes direcciones web:

En Español:

- http://www.publimas.com/tutoriales/vrml/documentos/inicio.htm
  Fecha último ingreso: 2002-09-12
- http://www.activamente.com
  Fecha último ingreso: 2002-09-12

En Inglés:

- http://www.cosmosoftware.com
  Fecha último ingreso: 2002-09-12
- http://www.vrml.org/VRML2.0/FINAL/
  Fecha último ingreso: 2002-09-12

Libros:

- Kris Jamsa, Phil Schmauder y Nelson Yee. "VRML Biblioteca del programador", Ed. McGraw – Hill, Primera Edición, 1998.